Resilient and Scalable Android Malware Fingerprinting and Detection

ElMouatez Billah Karbab

A Thesis

in

The Concordia Institute

for

Information Systems Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy (Information and Systems Engineering) at

Concordia University

Montréal, Québec, Canada

April 2020

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Mr. ElMouatez Billah Karbab

Entitled: Resilient and Scalable Android Malware Fingerprinting and Detection

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Information and Systems Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Akshay Kumar Rathore

_____ External Examiner
Dr. Ali Miri

_____ External to Program
Dr. Emad Shihab

_____ Examiner
Dr. Amr Youssef

_____ Examiner
Dr. Lingyu Wang

_____ Supervisor
Dr. Mourad Debbabi

Approved by _____
Dr. Abdessamad Ben Hamza, Director
Concordia Institute for Information Systems Engineering

April 1st, 2020
Date of Defence

_____
Dr. Amir Asif, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Resilient and Scalable Android Malware Fingerprinting and Detection

ElMouatez Billah Karbab, Ph.D.

Concordia University, 2020

Malicious software (Malware) proliferation reaches hundreds of thousands daily. The manual analysis of such a large volume of malware is daunting and time-consuming. The diversity of targeted systems in terms of architecture and platforms compounds the challenges of Android malware detection and malware in general. This highlights the need to design and implement new scalable and robust methods, techniques, and tools to detect Android malware. In this thesis, we develop a malware fingerprinting framework to cover accurate Android malware detection and family attribution. In this context, we emphasize the following: (i) the scalability over a large malware corpus; (ii) the resiliency to common obfuscation techniques; (iii) the portability over different platforms and architectures.

In the context of bulk and offline detection on the laboratory/vendor level: First, we propose an approximate fingerprinting technique for Android packaging that captures the underlying static structure of the Android apps. We also propose a malware clustering framework on top of this fingerprinting technique to perform unsupervised malware detection and grouping by building and partitioning a similarity network of malicious apps. Second, we propose an approximate fingerprinting technique for Android malware's behavior reports generated using dynamic analyses leveraging natural language processing techniques. Based on this fingerprinting technique, we propose a portable malware detection and family threat attribution framework employing supervised machine learning techniques. Third, we design an automatic framework to produce intelligence about the underlying malicious cyber-infrastructures of Android malware. We leverage graph analysis techniques to generate relevant, actionable, and granular intelligence that can be used to identify the

threat effects induced by malicious Internet activity associated to Android malicious apps.

In the context of the single app and online detection on the mobile device level, we further propose the following: Fourth, we design a portable and effective Android malware detection system that is suitable for deployment on mobile and resource constrained devices, using machine learning classification on raw method call sequences. Fifth, we elaborate a framework for Android malware detection that is resilient to common code obfuscation techniques and adaptive to operating systems and malware change overtime, using natural language processing and deep learning techniques.

We also evaluate the portability of the proposed techniques and methods beyond Android platform malware, as follows: Sixth, we leverage the previously elaborated techniques to build a framework for cross-platform ransomware fingerprinting relying on raw hybrid features in conjunction with advanced deep learning techniques.

Dedication

I dedicate this thesis to my family, Said, Aziza, Battoul, Yosr, Mozn, Yomn, Gyhm, Hamza, Ahlem, Manar, Montaha, Hamza, Abdullah, Nossaiba, Rima, Lahcen, and Souhila. Without your support and love, I would not able to finish this work.

Acknowledgments

I would like to express my sincere gratitude to my thesis supervisor Dr. Mourad Debbabi for his tremendous and exceptional support during my PhD. I am grateful for his mentorship on cyber security and related applications and also for his guidance on how to think and behave like a researcher. Indeed, I have learned from you many lessons during my PhD that transformed me from an eager student to a competent researcher. Without your help and support, Dr. Debbabi, I would not have completed my thesis, thank you.

I would also like to express my gratitude to Dr. Miri for his kind acceptance to evaluate my thesis as external examiner and for the numerous comments and suggestions that significantly enhanced the quality of the thesis. I would also like to extend my thanks to the thesis examiners Dr. Shihab, Dr. Youssef, and Dr. Wang for their contribution to the thesis through their active role in the supervision committee. I would like to express my gratitude to the professors that I learned from them during my PhD: Dr. Dssouli, Dr. Wang, Dr. Youssef, Dr. Shihab, Dr. Miri, Dr. Mannan, Dr. Clark, Dr. Chadi, Dr. Derhab, Dr. Djenouri, Dr. Aliouat, Dr. Hanna, and Dr. Mokhov.

My gratitude goes also to my colleagues, Djedjiga Mouheb, Amine Boukhtouta, Saed Alrabaee, Sujoy Ray, Andrei Soeanu, Paria Shirani, Perry Jones, Houssem Eddine Bordjiba, Quentin Le Sceller, Abdullah AL-Barakati, Sadegh Torabi, Elias Bou-Harb, Claude Fachkha, Aniss Chohra, Taous Madi, Badis Racherache, Abdullah Qasem, Gaby Dagher, Onur Duman, Pratyusha Bhattacharya, and He Huang, for their collaboration and friendship during all stages of my PhD. Thank you Dr. Andrei Soeanu for your insightful comments. A special thanks goes to my friend and brother Nour-Eddine Lakhdari for all his help and support. I would also like to thank the staff of the Concordia Institute for Information System Engineering (CIISE) for their help during my PhD.

I would especially like to thank my wife Battoul for keeping the home fires burning as I was away in my office over the last years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mobile apps have become an inherent part of our everyday life, as many of the services we use are provided to us through mobile apps. Mobile apps changed the way we communicate and put smart devices in the center of many of our daily activities. In contrast to personal computers (non-mobile systems), smart devices are equipped with sophisticated sensors, from cameras and microphones to gyroscopes and GPS [98]. These various sensors enable a whole new world of applications for end-users [98] and generate vast amounts of data, which contain highly sensitive information. This raises the need for security solutions to protect users from malicious apps, which exploit the sophistication and the sensitive content of smart devices.

Android OS witnessed a phenomenal growth, being deployed on wide spectrum of smart devices. It has the most significant share in the mobile computing industry with 76.1% in 2019-Q3 [45] due to its open-source distribution and appealing features. Besides, it has become not only the dominant platform for mobile phones and tablets but is also gaining increasing attention and penetration in the Internet of Things (IoT) realm [2, 11, 25, 44]. In this context, Google has launched Android Things [58], an Android OS for IoT devices, where developers benefit from the mature Android stack to develop IoT apps targeting thin devices [1, 25, 38, 58]. In this context, protecting Android devices from malicious apps is of paramount importance. The sophistication of mobile devices and the ubiquitousness of IoT objects help to build a smart world, but also unleash unprecedented potential for cyber-threats. Such threats could be committed by adversaries who might gain access to sensitive information and resources through Android malware apps.

1.1 Motivations

The volume of malware is growing tremendously [29], millions per month. In 2013, there was about 30 Million malware in the whole year [36]. In only February 2017, the number of new malware variants has reached to 94.6 Million [56]. This phenomenal growth is due to the ease of development of malicious apps nowadays, especially, repackaging existing malicious apps to develop new variants. There are some discrepancies in the estimation of the actual daily new malware [36, 56], but they agree to a large extent. In 2013, the estimation on new malware was about 82k per day [36]. However, in 2017, experts [41] estimated a figure of 250k new malware a day. The challenge relates to processing, analyzing, and fingerprinting new malware binaries to produce analytics in a limited time window. In this respect, our research aims at answering the following questions: (i) How can we efficiently fingerprint malware in large binary corpus? (ii) How can we effectively detect malware? (iii) How can we group the detected samples into malware families?

The popularity of Android OS and its open nature make it a tempting target for malicious apps in mobile computing platforms [179], including mobile devices (e.g., phones and tablets) as well as IoT devices [4, 55]. The reported Android malicious apps increased by 40% in the third quarter of both 2017 and 2018 [54]. Android malicious apps target billions of users through centralized app markets, where developers publish end-user apps. The simplicity and affordability of malicious apps deployment in various app stores increase the number of compromised devices. Many security solutions have been proposed to defend against Android malware. For example, the vetting system in app markets such as Google Play[1] plays a crucial role in detecting Android malware. However, Android malicious apps are able to deceive the vetting systems [2] in app markets[3] and infect millions of devices[4], which emphasizes the importance of effective Android malware detection capability.

There is a clear need for solutions that defend against malicious apps in mobile and IoT devices with specific requirements to overcome the limitations of existing Android malware detection systems. First, the Android malware detection system should ensure a high accuracy with minimum

[1] https://play.google.com
[2] https://tinyurl.com/y57twlbs
[3] https://tinyurl.com/yx8vqld2
[4] https://tinyurl.com/y5rwybst

false alarms. Second, it should be able to operate at different deployment scales: (i) Server machines, (ii) Personal computers, (iii) Smartphones and tablets, and (iv) IoT devices. Third, detecting that a given app is malicious may not be enough, as more information about the threat is required to prioritize the mitigation actions. Knowledge on the type of attack could be crucial to prevent the intended damage. Therefore, it is essential to have a solution that goes a step further and attributes the malware to a specific family, which defines the potential threat that infected system is exposed to. Finally, it is necessary to minimize manual human intervention as much as possible and make the detection dependent mainly on the app sample for automatic feature extraction and pattern recognition. As malicious apps are quickly getting stealthier, the security analyst should be able to catch up with this trend. In this respect, for every new malware family, a manual analysis of the samples is required to identify its pattern and features that distinguish it from benign apps.

1.2 Research Objectives

In this thesis, we focus on fingerprinting malware, which is the process of finding distinctive patterns in malware in order to be distinguished from benign samples, as a core procedure in the malware detection and family attribution. The main goals of the thesis are as follows:

(1) **Android Malware Fingerprinting**: Malware detection and family attribution of Android malicious apps is a primary goal of this thesis. The developed frameworks and techniques are applied to Android malware. However, the framework is general enough to be accommodated to address malware on other platforms.

(2) **Ransomware Fingerprinting**: For ransomware detection, we apply the learned techniques on the ransomware detection problem. In this context, we target the problem of cross-platform ransomware detection.

Toward achieving the aforementioned goals, we dedicate special attention to the following criteria:

(1) **Scalable Malware Fingerprinting**: In response to the increasing number and magnitude of malware attacks, in this thesis, we focus on proposing scalable solutions and techniques for malware fingerprinting, while maintaining high malware detection performance.

(2) **Resilient Malware Fingerprinting**: Malware developers employ various obfuscation techniques to thwart detection attempts. Therefore, obfuscation resiliency is crucial in modern malware fingerprinting. In this thesis, we put the emphasis on resiliency to code transformation and common obfuscation techniques in the development of our malware fingerprinting techniques and systems.
(3) **Portable Malware Fingerprinting**: (i) Manual feature engineering of platform-dependent malware features is not scalable given the amount and the changing velocity of malware techniques. Therefore, portable feature engineering is an essential criterion in the development of fingerprint solutions, such as feature engineering in dynamic analysis. (ii) The efficiency of the fingerprinting techniques, starting from the initial data processing to the detection, is a crucial factor that affects the deployment portability of the solution. (iii) The diversity of targeted architectures, platforms, and execution environments of malware is a challenging problem. The portability of malware fingerprinting tools across different architectures (ARM, x86, MIPS, etc.) is an important requirement of the fingerprinting process.

1.3 Research Contributions

In this section, we present a summary of the thesis contributions.

1.3.1 Community-Based Android Malware Clustering

In this contribution, we propose APK-DNA [133], a fuzzy fingerprinting approach that captures both the structure and the semantics of the *APK* file using most Android *APK* features. Using APK-DNA, we propose the Cypider [130] framework, a set of techniques and tools aiming to perform a systematic detection and grouping of mobile malware (Android Malware) by building an efficient and a scalable similarity network of malicious apps. Our detection method is based on a novel concept, namely *malicious community*, in which we consider, for a given family, the instances that share common features. Under this concept, we assume that multiple similar Android apps with different authors are most likely to be malicious. Cypider leverages this assumption for the detection of variants of known malware families and zero-day malware. Cypider applies *community detection algorithms* on the similarity network, which extracts sub-graphs considered as

suspicious and most likely malicious communities. We propose a novel fingerprinting technique, namely *community fingerprint*, based on a learning model for each malicious community.

1.3.2 Android Malware Fingerprinting on Dynamic Analysis

In this contribution, we propose DySign, a novel technique for fingerprinting Android malware's dynamic behaviors. DySign allows for the generation of a digest from the *dynamic analysis* of a malware sample. On top of DySign [128, 129], we propose, MalDy [125, 127], a portable malware detection and family threat attribution framework using supervised machine learning techniques. The key idea of MalDy portability is the modeling of the behavioral reports as a sequence of words, along with advanced Natural Language Processing (NLP) and Machine Learning (ML) techniques for automatic engineering of relevant security features to detect and attribute malware without human in the loop. More precisely, we propose to use the *bag-of-words* (BoW) model to capture the behavioral reports. Afterward, we build ML ensembles on top of BoW features. We evaluate MalDy on various datasets from different platforms (Android and Win32) and execution environments. The evaluation shows the effectiveness and the portability of MalDy across a spectrum of analyses and settings.

1.3.3 Fingerprinting Android Malware Cyber-Infrastructure

In this contribution, we present ToGather [124, 126], an automatic investigation framework that takes Android malware samples as input and produces insights about the underlying malicious cyber-infrastructures. ToGather leverages state-of-the-art graph analyses techniques to generate actionable, relevant, and granular intelligence to detect the threat effects induced by the malicious Internet activity of Android malware apps. The main contributions are: (1) We design and implement of ToGather, a simple, yet practical framework for the generation of actionable, relevant and granular intelligence on the malicious cyber-infrastructures used by Android malware. (2) We propose a correlation mechanism with multiple cyber-threat intelligence feeds, which enriches not only the resulting malicious cyber-infrastructure intelligence but also the labeling of the tracked malicious activities.

1.3.4 Portable Supervised Android Malware Detection

In this contribution, we propose MalDozer [69, 131, 132], an automatic Android malware detection, and family attribution framework that relies on method call sequence classification using deep learning techniques. Starting from a raw sequence of the app's API method calls, MalDozer automatically extracts and learns the malicious and the benign patterns from the actual samples in order to detect Android malware. MalDozer can serve as a ubiquitous malware detection system that can be deployed not only on servers, but also on mobile and even IoT devices. This framework has the following main contributions: (1) MalDozer, a novel, effective, and efficient Android malware detection framework using the raw sequences of API method calls in conjunction with neural networks. In this context, we take a step beyond malware detection by attributing the detected Android malware to its family with high accuracy. (2) We propose an automatic feature extraction technique during the training using *method embedding*, where the input is the raw sequence of API method calls, extracted from Android Dalvik assembly.

1.3.5 Robust and Adaptive Android Malware Fingerprinting

In this contribution, we propose PetaDroid, a resilient and adaptive framework for android malware detection and family clustering using advanced natural language processing and machine learning techniques. PetaDroid detects Android malware samples using an ensemble of Convolutional Neural Networks (CNNs) on top of our Inst2Vec features. Afterward, PetaDroid clusters the detected malware into groups of the same family utilizing sample digests generated using deep neural auto-encoder. PetaDroid is robust to common obfuscation techniques due to our fragment randomization technique during the training. PetaDroid leverages the confidence of detection decisions during deployment to collect extension dataset at each epoch. The extension dataset is used to automatically build new models without manual sample collection and also to empower time resiliency.

1.3.6 Ransomware Fingerprinting using Hybrid Features

In this contribution, we propose SwiftR, a cross-platform framework for ransomware fingerprinting. SwiftR provides fast and accurate ransomware detection that relies on raw hybrid features with advanced deep learning techniques. SwiftR proposes a novel ransomware detection framework that is agnostic to architectures and platforms by introducing the use of intermediate representation (IR) features from static analyses. Also, SwiftR proposes a novel ransomware detection using word-based features from behavioral analysis reports produced by dynamic analyses. We endow SwiftR with a novel neural network architecture that leverages the different contents of the malicious sample to identify ransomware. In this framework, we make the following contributions: (1) We propose a novel cross-platform *ransomware* fingerprinting framework. SwiftR can detect ransomware, distinguish it from general malware, and attribute it to a ransomware family. (2) *Static SwiftR:* We propose novel implicit static analysis features extracted from VEX intermediate representation (IR). Furthermore, we design a novel multi-task hierarchical neural network for learning relevant ransomware patterns from these raw static features. (3) *Dynamic SwiftR:* We proposed novel word-based features and LSTM neural network machine learning to achieve portable ransomware fingerprinting.

1.4 Thesis Organization

The remainder of the thesis is organized as follows: Chapter 2 provides the necessary background and state-of-the-art to Android malware detection. In Chapter 3, we propose an approximate fingerprinting approach for malware detection and apply it to Android malware. Afterward, we present a novel malware clustering technique, based on the aforementioned fingerprinting approach and graph partition techniques. Chapter 4 tackles portable malware fingerprinting from dynamic analysis reports using natural language processing and machine learning techniques. In Chapter 5, we propose a framework for Android malware cyber-infrastructure investigation. In Chapter 6, we propose a portable Android malware detection using deep learning techniques. In Chapter 7, we present an Android malware detection framework with high obfuscation resiliency and change overtime adaptation. Chapter 8 presents a ransomware fingerprinting framework that applies the elaborated techniques from the previous chapters of the thesis. Finally, Chapter 9 provides

the conclusions, which discuss the relevance and importance of the addressed problems, followed by a summary of our contributions. In addition, it mentions the limitations of our research while presenting some avenues for future research on the topics studied.

Chapter 2

Background and Related Work

In this chapter, we review and compare state-of-the-art proposals on Android malware analysis and detection according to a novel taxonomy. Due to the large number of published contributions, we focus our review on the most prominent articles in terms of novelty and contributions, with an emphasis on those published in top-tier security journals and conferences. The proposed taxonomy is based on the generality of Android malware threats. It classifies the existing systems into: (1) *general malware detection*, which aims to detect malware without taking into account a particular type of attack, and (2) *attack-based malware detection*, which aims at detecting specific attacks such as privilege escalation attacks, data leakage attacks, etc. Furthermore, each threat category is classified according to the system deployment of the detection approach, i.e., the physical environment into which the system is intended to run. Furthermore, we consider three main deployment architectures: *workstation-based*, *mobile-based*, and *hybrid* architectures. The proposed two-level taxonomy allows carrying out an objective and appropriate analysis by comparing only systems that are addressing the same threat category, and having the same deployment architecture as they share the same goals and have similar issues to solve.

2.1 Background

In this section, we introduce the essential background knowledge of Android OS. We also discuss briefly Android security and its implication on malware detection.

2.1.1 Android OS Overview

Android is a mobile operating system maintained by Google and supported by the Open Handset Alliance (OHA) [7]. Android is embraced by the Original Equipment Manufacturers (OEMs), chip-makers, carriers and application developers. Android apps are written in Java. However, the native code and shared libraries are developed in C/C++ [19]. The current Android architecture [39] consists of the Linux kernel, which is designed for an embedded environment with limited resources. On top of the Linux kernel, there is the Hardware Abstraction Layer (HAL), which provides standard interfaces that expose device hardware capabilities to the higher-level Java API framework, by allowing programmers to create software hooks between the Android platform stack and the hardware. Also, there is Android Runtime (ART), which is an application runtime environment used by the Android OS and which replaced Dalvik virtual machine starting from Android 5.0. ART translates the apps' bytecode into native instructions that are later executed by the device's runtime environment. ART introduces the Ahead-Of-Time (AOT) compilation feature, which allows compiling entire applications into native machine code upon their installation. The native libraries developed in C/C++ support high-performance third-party reusable shared libraries. The Java API framework provides APIs for the building blocks the user needs to create Android apps.

Android Apk Format

Android Application Package (*Apk*) is the file format adopted by Android for apps distribution and installation. It comes as a *ZIP* archive file, which contains all the components needed to run the app. By analogy, *Apk* files are similar to Windows *EXE* installation files or Linux *RPM*/*DEB* files. The *Apk* package is organized into different directories (namely **lib**, **res**, **assets**), and files (namely **AndroidManifest.xml** and **classes.dex**). More precisely, the **AndroidManifest.xml** file contains the app meta-data, e.g., name, version, required permissions, and used libraries. The **classes.dex** file contains the compiled Java classes. The **lib** directory stores C/C++ native libraries [19]. The resources directory (**res**) contains the non-source code files, such as video, image, and audio files, which are packaged during compilation.

Android Markets

Android app market is an Internet site for developers to publish their apps. Google Play is the official app market for Android. Before an app is published in this market, it needs to be verified by the Bouncer vetting system [155] to check newly submitted apps against malware. This involves scanning an app for known malicious code and performing dynamic analysis for a limited period to detect hidden malicious behaviors. The vetting system can be evaded by apps that avoid triggering malicious behavior during the analysis time. There are also other third-party markets such as AppChina [40] and Mumayi [23], where developers can upload their apps. However, they provide fewer restrictions to publish apps. Unlike Google Play store, they do not vet the submitted apps but rather rely on users' feedback, which helps attackers to publish repackaged apps and malware easily (minimum vetting).

2.1.2 Android Security

Android OS employs two security mechanisms: permissions and sandboxing. In Android, apps can access resources such as telephony, network, and SMS functions using APIs. Android APIs are protected using a security mechanism based on permissions. Each application must define the permissions it requests in its AndroidManifest.xml file. A user needs to grant the required permissions to install the app. Otherwise, the application cannot be installed. The Android kernel provides a sandboxing feature, which isolates apps from one another. In Android, each application is assigned a unique User ID (UID) and is run as a separate process. The file system access policy does not allow one user (resp., application) to access or modify another user's (application's) files.

Android Security Threats

Attackers can exploit many weaknesses and vulnerabilities in the Android ecosystem to compromise and infect Android devices with malware. These weaknesses are summarized as follows: (1) Most of the App markets deploy no or limited vetting system to check whether the submitted apps are malicious or not. (2) The official Android market might contain malicious apps. Some apps [3, 10, 22] in Google's Play Store have been identified as malicious. Thus, there is a risk that users

download malware. (3) There are non-market sources to obtain Android apps such as SD cards, which open more attack entry points for malware. (4) Most of the users ignore or have little understanding of the Android permission policy. This is stated in a survey [110], which showed that only 17% of users look at the permissions when installing applications. (5) It has also been noticed that developers request more permissions than they need [109]. As legitimate and malicious applications can request permissions, it is often difficult for users to determine, during the installation time whether the requested permissions are harmful or not. (6) It is relatively easy for an adversary to reverse-engineer a legitimate Android app, insert malicious code, and repackage the Apk file again.

Design Challenges of Malware Detection Systems

The design of Android malware detection system faces many challenges: (1) Ensuring simultaneously high-accuracy detection and efficiency in terms of time and resource use (CPU, RAM, and battery in case of mobile device deployment), is difficult to achieve, especially in the case of deploying the detection system on resource-constrained devices. (2) Malware developers employ techniques to evade detection, such as code obfuscation or dynamically loading a binary code from a remote server. (3) The vetting system executes apps for a limited time in a controlled environment, e.g., sandboxing or emulation, to check their maliciousness. However, some malware only reveal their malicious behaviors after a period of time to escape the analysis of maliciousness detection systems during the early execution period. Also, some malware try to check the execution environment for signs of a sand-boxing system [128, 129]. The goal is to prevent the execution of the malicious payload under a sand-boxing environment.

2.2 Android Malware Detection Overview

In this section, we review the existing contributions on Android malware analysis. Android malware detection methods focus on identifying whether the analyzed app is benign or malicious. Malware detection proposals can be categorized into static [73, 111], dynamic [64, 87], and hybrid [81, 197] analysis-based.

Static Analysis Approaches: Static analysis techniques perform fast code disassembling and

decompilation without the need to execute the binary. The static methods [66, 73, 77, 86, 101, 111, 119, 133, 137, 157, 167, 192, 194, 196, 200] depend on static features, which are extracted from the Apk file such as requested permissions, APIs, bytecodes, opcodes. Static analysis techniques are fast and covers all the execution paths of the analyzed app. However, this approach is undermined by the use of various code transformation techniques [106]. We may divide static analysis based techniques into the following categories: (i) **Signature-based analysis:** This analysis deals with the extracted syntactic pattern features. The authors in [111] create a unique signature matching a particular malware. However, such signature cannot handle new variants of existing known malware. Moreover, the signature database should be updated to handle new variants. AndroSimilar [107] has been proposed to detect zero-day variants of known malware. It is an automated statistical feature signature-based method for malware detection. (ii) **Resource-based analysis:** The manifest file contains important meta-data about the components, i.e., activities, services, receivers, etc. and the required permissions. There are some methods that have been proposed to extract such information and subject it to analysis [92, 111, 112, 133]. (iii) **Permission-based analysis:** This approach is based on discovering unnecessary permission requests that might lead to malicious activity [78, 169]. In [104], the authors proposed a certification tool that defines a set of rules to detect malware by identifying combinations of requested permissions. (iv) **Semantic-based analysis:** There are existing approaches that analyze Dalvik bytecode that is semantically rich, containing type information such as classes, methods and instructions. Additionally, such information can be used to analyze control and data flow graphs that reveal privacy leakage and telephony services misuse [114, 114, 135, 135].

Dynamic Analysis Approaches: The dynamic methods [64, 71, 87, 118, 168, 186, 199] use features that are derived from the app's execution. They are more resilient to code obfuscations than static analysis methods. However, such methods [85, 97, 102, 103, 118, 163, 172] incur additional cost in terms of processing and memory to run the app. Also, anti-emulation techniques such as sandbox detection and delaying malware execution can evade dynamic analysis methods. Dynamic techniques are divided into the following two categories: (i) **Resources usage based:** Some malicious apps may cause Denial of Service (DoS) attacks by over-utilizing constrained hardware resources. A range of parameters such as CPU usage, memory utilization statistics, network traffic

pattern, battery usage and system-calls for benign and malware apps are gathered from the Android subsystem. Then, automatic analysis techniques along with machine learning techniques are employed [97, 163, 172]. (ii) **Malicious behavior based:** This is related to abnormal behaviors such as sensitive data leakage and sending SMS/emails [85, 97, 102, 103, 118, 163, 172].

Hybrid Analysis Approaches: The hybrid methods [65, 81, 114, 121, 145, 150, 175, 184, 197, 204] use both static and dynamic features.

Malware Family Attribution: In the previous categories, the proposed systems focus mainly on the detection task in which we segregate malware and benign apps. In this category, the proposed systems focus on the malware family attribution task, in addition to the detection task, as a goal for the analysis. Malware family attribution aims to attribute the malware to its actual family. To secure Android systems, some methods [63, 99, 143] focus on detecting variants of known families. Other proposals [88, 94, 95, 105, 108, 115, 130, 136, 144, 178, 180, 183, 201, 202] adopt the unsupervised learning approach to find families of similar apps. These proposals assume that two or multiple apps that share similar code are likely to belong to the same malware family. Thus, they check if the apps are using the similar malicious code (i.e., detection of malware families), or they check for reused code of the same original app (i.e., code reuse detection).

2.3 Taxonomy of Android Malware Detection Systems

In this section, we present our taxonomy for Android malware detection systems. As shown in Figure 2.1, the taxonomy considers three aspects for the classification and the comparative study: (1) *Targeted threat* , (2) *System deployment*, and (3) *Android stack layer* aspects are for classification. The first three aspects, i.e., targeted threat, system deployment, and the implementation layer are used to cluster related works that address the same threats in the same performance objectives (i.e., they operate under the same class of deployment settings); finally it defines on which Android OS stack layer, the system is implemented.(i.e, *app*, *framework*, and *Linux kernel* layer as shown in Figure 2.3). Afterward, we conduct a comparative study on the resulting groups for the criteria of the last two aspects, i.e., Feature Selection and Detection strategy.

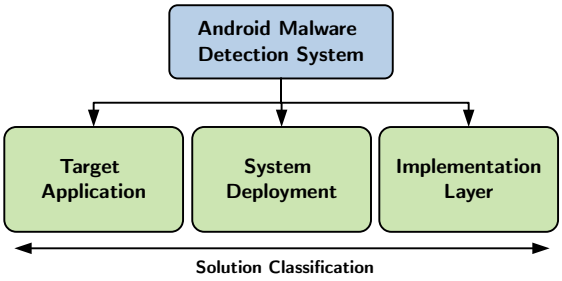


Figure 2.1: Classification Aspects

2.3.1 Malware Threats

Android Malware detection systems are designed to protect against different kinds of malicious activities. Based on the targeted threats, we classify them, as shown in Figure 2.2, into: *Attack-based detection systems*, *generic detection systems*, and *helpers systems*. By (1) *attack-based detection systems* (also attack-dependent systems), we mean the systems that aim to address specific attacks. For instance, *Graphical User Interface (GUI) phishing* in which the adversary might develop a malicious app whose graphical user interface (GUI) is visually similar to a legitimate app, to deceive the user. In another example, two or more apps can collaborate to launch a *collusion attack* leveraging Android Inter-Components Communication (ICC). Although each app has a set of non-critical permissions, the apps can collude to generate a joint set of permissions that enables them to perform unauthorized malicious activities.

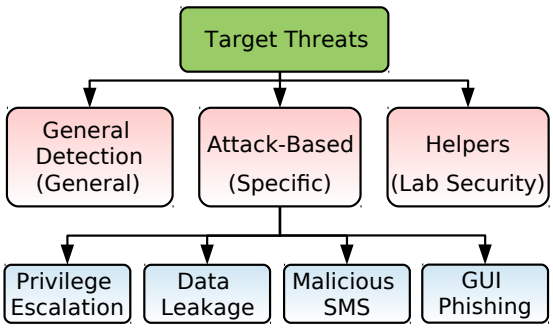


Figure 2.2: Classification of target threats

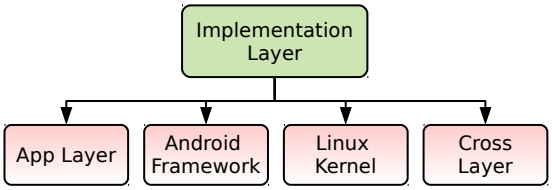


Figure 2.3: Classification of the Implementation Layer

By *generic detection systems*, we mean systems that aim to detect malware without considering specific attacks. Instead, they detect malware based on prior knowledge of specific malware patterns

or a learned model. Alternatively, other detection systems focus on apps that contain similar code or exhibit similar behaviors.

Helpers systems provide assistance to Android malware detection solutions. First, they are mainly used in lab environments due to their need for heavy manual work. Second, the provided tool or results of such assets will be leveraged to improve and enhance the actual malware detection solutions such as code obfuscation tools. Finally, these assets may not have been directly related to malware detection, but they could help sharpen the malware detection rate.

2.3.2 Detection System Deployment

Detection system deployment refers to the adopted architecture in terms of physical components. An Android malware detection system can be deployed on different types of architectures: (i) *workstation*, (ii) *Mobile*, and (iii) *hybrid*, as shown in Figure 2.4. Each type of architecture has its design objectives and operates under specific constraints and deployment settings, as described below:

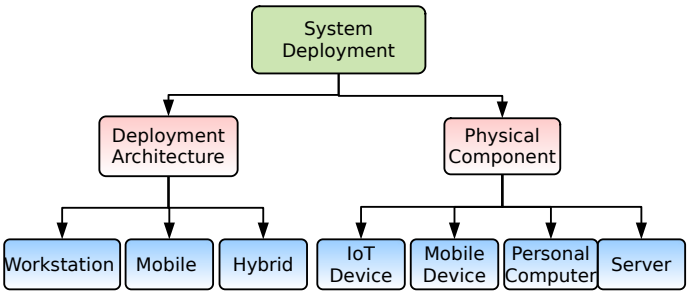
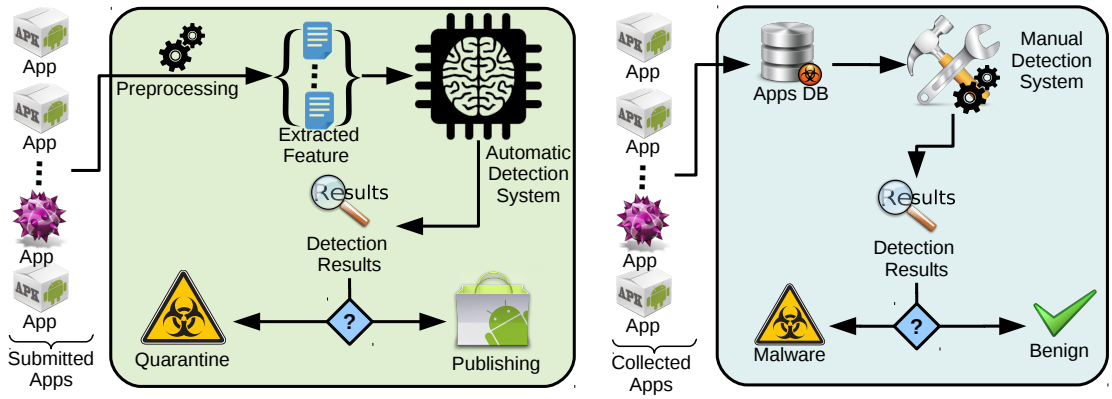


Figure 2.4: System Deployment Classification

Workstation-based architecture: This type of architecture is centralized because all the detection modules are deployed on a high-power server or high-end desktop machine, and it can be used for two types of application scenarios:

First, *App market analysis*, in which the detection system, as depicted in Figure 2.5(a), has to check a newly submitted app before publishing it on the market. It represents the first line of defense in the Android ecosystem. For this reason, high detection accuracy is required. The second requirement is the scalability of the detection system with respect to detection time to the high arrival rate of apps, and hence the detection should be performed in real-time, i.e., online detection



(a) App Market Analysis (b) Security Lab Analysis

Figure 2.5: Workstation-Based Analysis

mode.

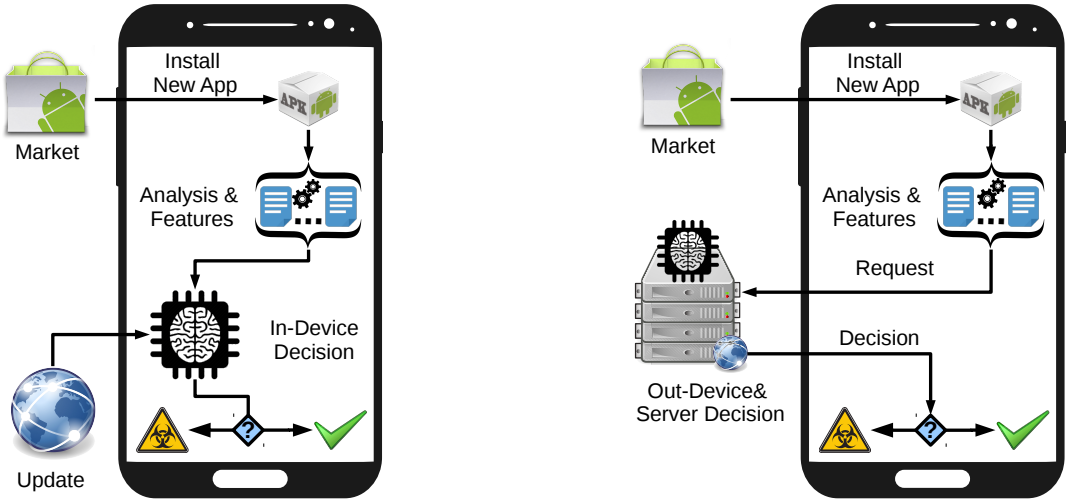
Second, *Security lab analysis*, the security lab analyst can be any user working in the industry, a researcher from academia, or independent analyst, who uses machines of middle-resource capability regarding CPU and memory such as desktop computers and laptops, as depicted in Figure 2.5(b). We need this separate category because the malware analysis may involve manual investigation by the security practitioner; in contrast, app market analysis category has fully automated detection process. Thus, it is important to stress that this category leverages experts knowledge and may involve manual analysis.

Mobile-based architecture: Any user with a mobile device can perform this analysis. The detection operations (provided by preprocessing, feature extraction, and the detection components) are carried out on the mobile device, as depicted in Figure 2.6(a). In addition to ensuring high detection accuracy, the detection system should cope with the limited capabilities of mobile devices that are characterized by constrained resources such as CPU, memory, and battery capacity.

Hybrid architecture: If the malware detection system operations are split between the mobile device and a workstation, the architecture is called hybrid (see Figure 2.6(b)). In this case, the system should incur low communication cost in addition to consuming limited resources in terms of CPU, memory, and battery.

As each type of deployment system has different priority design objectives and offers different

resource capacities from one another, we compare the detection systems that are categorized under the same deployment type. The differences between deployment systems regarding the importance of design objectives are summarized in Table 2.1.



(a) Moblie-Based Detection (b) Hybrid Detection

Figure 2.6: Hybrid and Mobile-Based Analysis

Architecture	Workstation-Based		Mobile-Based	Hybrid
Deployment	App Market Analysis	Security Lab Analysis		
High Accuracy	Very Important	Important	Very Important	Very Important
Detection Time	Very Important	Important	Important	Important
CPU	Less important	Important	Very important	Very important
RAM	Less important	Important	Very important	Very important
Battery	Not important	Not Important	Very important	Very important
Communication	Not important	Not Important	Not Important	Very important
Usability	Not important	Not important	Important	Important

Table 2.1: Design Objectives of Deployment Systems

2.4 Performance Criteria for Malware Detection

In this section, we present a generic framework as a template for Android malware detection systems together with the performance criteria. This template helps to make an abstraction of the components of a typical Android malware detection system and its characteristics and criteria based on its position in the proposed taxonomy (previous section). The framework, as shown in Figure 2.7,

consists of two main processes: *Process to generate detection patterns*, which builds the detection pattern (or model), and *Detection and response process*, which analyzes a target Apk based on the developed detection model. In general, a dataset of Apks is the input for the Detection pattern generation process. The Apk is first processed through disassembling or decompilation to generate the raw features as they exist in the Apk file. In some cases, the raw features are processed to generate high-level features, which are, in turn, fed to training modules to produce the detection model. Given a target Apk, the detection module will determine whether it is malicious or benign. The detection result can take one of the following two forms: label (i.e., malicious or not) and score (i.e., risk score). Also, a detection response might follow after obtaining the detection result.

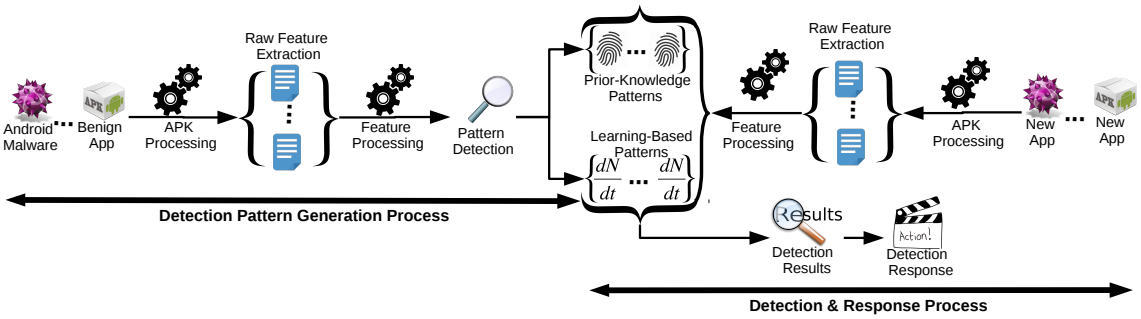


Figure 2.7: Template Framework for Android Malware Detection Systems

2.4.1 Feature Selection

As shown in Figure 2.8, features can be (1) Static, extracted from the Apk file, such as, permissions, API used, opcode, etc. (2) Dynamic, which is extracted from the running the app, such as system calls, invoked APIs, network traffic, etc. (3) Hybrid, which combines both static and dynamic features. The features can also be classified with respect to other aspects as follows:

Code transformation and Obfuscation resiliency: An obfuscation technique aims to evade detection by instrumenting the features used by the detection model. The feature is said to be highly resilient to an obfuscation technique if the malware detection process using such a feature is less affected by such technique, or the malware needs to change its functional logic to evade detection.

Adaptation to OS and malware evolution: The release of a new Android OS version implies a new set of API frameworks. Detection systems that consider APIs as features need to manually redefine

the set of API features before applying the detection process, which makes the adaptation of the detection system difficult.

Features preprocessing complexity: In static analysis, before extracting some raw features (APIs used, opcode), it is required to disassemble the dex file, which takes a longer time than removing permissions, which only requires accessing the Manifest file. Although permissions are quickly extracted, the detection methods mainly based on such features are less resilient to obfuscations compared to those employing features that are derived from the dex file.

Detection required runtime: This measures the time interval between processing the Apk and making a decision (malware or not) about a given Android app.

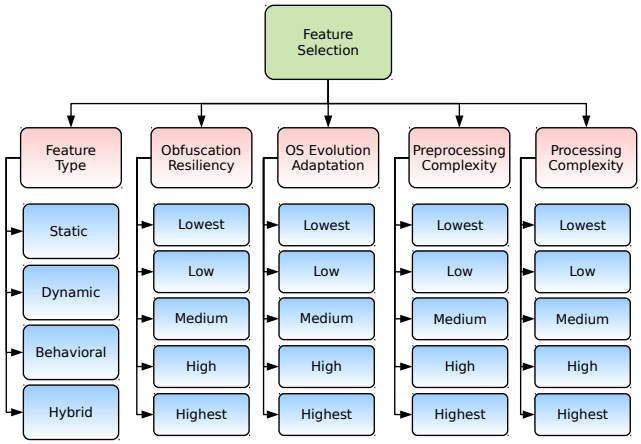


Figure 2.8: Feature Selection Criteria

2.4.2 Detection Strategy

As shown in Figure 2.9, the detection can be either offline or online. Upon malware detection, the *detection response* can be either: passive notification sent to the user, active reaction, which blocks the malware, or no response in the case of App market analysis and security lab analysis. The *detection scope* defines the goals of the android malware fingerprinting: only malware detection, family attribution, or go a step further such as the detection of threat network, composed of IP addresses and domains names, related to Android malware samples. The *detection approach* indicates how the detection model is produced, which can be through a learning procedure or a prior-knowledge. The prior-knowledge models are specification-based models that are manually

constructed by a human expert based on rules that try to determine the legitimate or the malicious behavior of the app. The main advantage of the prior-knowledge techniques is short detection as they only check if the predefined rules are violated or not. The main drawback is that building knowledge requires high-level human expertise and is often time-consuming and a difficult task. As for the learning models, they are automatic and can adapt to changes when new information is acquired.

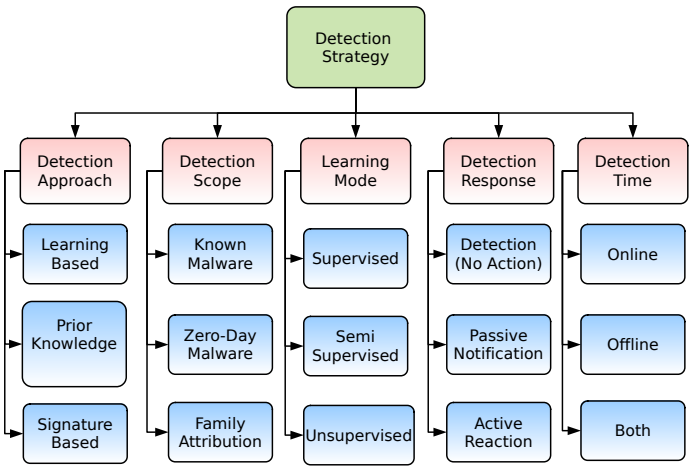


Figure 2.9: Detection Strategy Comparison Criteria

2.5 General Malware Threat Detection

In this section, we discuss Android malware detection proposals in the literature that target general malware detection without focusing on : *type of malware*, and *its attack techniques*. Moreover, this category leverages general features to fingerprint the malware. We classify the proposals in this category into *workstation-based*, *mobile-based*, and *hybrid-based* solutions. This section comes after we define our classification taxonomy and comparison criteria to describe and compare between the solutions in the light of our taxonomy and criteria.

2.5.1 Workstation-Based Solutions

This section presents the *workstation-based* solutions, which require a lot of resources in terms of processing and memory, so its deployment must be on relatively powerful machines.

MAMADROID [148, 158] is a framework for Android malware detection using static behavioral features of Android apps. MAMADROID leverages static analysis to extract learning features from a reverse engineered Android app, and Android APIs call sequences (a sequence of API is a possible behavior of the Android app). Markov chains are used to model and represent abstract forms of API sequences; the abstracted APIs consists of (i) Package such as java.lang and (ii) Family such as java. In this model, a node is an Android API, and an edge is the probability of having a transition from one API to another. The authors consider these probabilities as features for the MAMADROID machine learning system. Therefore, Markov chains play the role of features extractor, and it is the cornerstone of the proposed detection system. MAMADROID can only be deployed in large-scale and high-end machines due to the required resources. It is relatively slow due to intense preprocessing; the reported runtime might reach 13min. to 18min. depending on the size of the binary.

StormDroid [91] is a malware detection system that considers malware as a stream of apps as in the case of App markets. The authors are motivated by the fact that most existing systems rely mainly on permissions and sensitive Android APIs. Therefore, they propose, StromDroid as a machine learning system for Android malware detection. StromDroid leverages both static and dynamic analysis to achieve a high detection rate with a low false-positive rate. The evaluation of StromDroid includes multiple types of machine learning classifiers; K-Nearest Neighbors is the most accurate one. However, K-Nearest Neighbors is not sufficiently scalable because the detection runtime grows linearly with the size of the training dataset; but it is constant for classification algorithms.

The main goal of OpSeq [63] is to measure the similarity of unknown apps relative to known malware. OpSeq defines similarity as a function of normalized opcode sequences found in sensitive functional modules as well as app permission requests. OpSeq extracts the components from a known sample and creates corresponding signatures, which are used to search for similar components in target applications. OpSeq improves the use of opcode-sequence similarity by focusing only on the components that make suspicious API calls. The list of requested permissions is used as a second parameter to improve detection accuracy. OpSeq is tested on a dataset of $1,192$ known

malware samples belonging to 25 different families, 359 benign apps, and 207 new obfuscated malware variants. The results show that OpSeq can correctly detect known malware with an F1-Score of 98%. However, OpSeq generates fingerprints from only assembly opcodes N-grams from components that make suspicious API calls: (1) Relying on opcodes only ignores other information in the assembly code. (2) The system might not be scalable overtime due to the rapid change in suspicious API calls (how can find out new suspicious APIs in new OS versions).

2.5.2 Mobile-Based Solutions

This section presents *mobile-based* solutions. The following research initiatives target mobile deployment. In the mobile-based solutions, the design of the detection process (from the preprocessing, and feature extraction, to the final decision) is optimized to fit in resources-constrained devices with small memory capacity and computation power.

Drebin [73] is a supervised machine learning system for android malware detection. Static analysis is used to extract a variety of code and app features from Dalvik and Manifest file, respectively. As for the code features, the authors rely mainly on the used permissions and Android APIs (or lightweight features), which have been filtered to consider only suspicious APIs, such as dynamic load or cryptographic API. App features, extracted from Manifest file, are mainly the requested permissions, the app components and intents, and the requested hardware components. Drebin only considers lightweight features so that the preprocessing would be very fast. Lightweight features extraction is essential for Drebin system efficiency to run on mobile devices (also high-end servers). Drebin leverages bit-vector to represent the extracted features in order to train a Support Vector Machine (SVM) model off-line. Drebin [73] shows high detection performance with low false positive. In addition to detection results, Drebin provides explanatory details to the end-user in terms of suspicious features scores that affect the detection decision. However, the usage of lightweight features impact the resiliency of the system against common code obfuscation techniques. Also, Drebin relies on manual feature engineering that produces a static list of features, which is less resilient to OS change overtime (new features due to the new APIs, permissions, etc.).

DroidBarrier [68] is a runtime process authentication model for Android. DroidBarrier provides legitimate apps (that are considered as benign at the launch of the system) with security credentials,

which are used for authentication when the associated processes are created. Processes that do not have credentials fail to authenticate, and their corresponding applications are considered malicious. The main disadvantage of this approach is the need to determine which requests are initially legitimate.

A machine learning approach for a malware detection system that runs on Android devices is proposed in [166]. The authors model the detection system as a machine learning model, specifically as an anomaly detection system (only train the detection model on benign apps). To this end, they leverage a large number of available benign apps to train the anomaly detection model. Therefore, benign apps are considered as part of the normal area while everything else is considered as anomalies or malicious apps. The authors extract from the Android apps features such as permissions and Control Flow Graphs (CFG). A one-class support vector machine model, i.e., the anomaly detection model, is trained on the previous features offline. Afterward, the model is deployed on Android devices.

2.5.3 Hybrid Solutions

Hybrid solutions have two parts that are deployed respectively in mobile end-devices and the end-servers. Both ends work collaboratively to achieve detection of Android malware.

Crowdroid [85] is a detection system, which is composed of a lightweight app, that is installed on a mobile device and interacts with a centralized server to make detection decisions. The system, at the mobile device, monitors the Linux Kernel system calls of each app and sends the gathered information to a centralized server. The latter collects these calls from different users and applies a clustering algorithm to distinguish between benign and malicious apps. Although the reported detection performance of Crowdroid is perfect in some cases, the employed dataset is very small to the generalize the performance results, as shown in Table 2.3. Also, the usage of anomaly detection tends to generate a considerable amount of false positives compared with supervised classification.

DRACO, a hybrid Android malware detection system in [81], is proposed. In [81], the authors propose DRACO, which is a hybrid Android malware detection system. The system uses both dynamic and static analyses on two different ends: end-user smart devices and high-end servers, respectively. DRACO is hybrid because it is deployed on Android devices and servers simultaneously.

In Android devices, DRACO collects dynamic analysis features such as the CPU, memory usage, and file system accesses. On the server-side, DRACO conducts a static analysis to extract features such as Android API calls. Both sets of features are used to train a support vector machine model as the core of the DRACO system. Using end-user smart devices as a dynamic analysis system raises the issue of privacy and security of DRACO itself, whether at the client or the server-side. Also, the used features are straightforward and are extracted completely in nowadays devices; therefore, this opens the question of the need for the servers.

Monet [181] is a runtime fingerprinting system for Android malware detection. Monet's generated fingerprints capture the behavior of the installed apps in the Android devices overtime. The authors design and implement Monet, a system that is deployed on both Android devices (client) and the high-end servers (server). Monet client is implemented in the Android middle-ware framework and Linux kernel to capture the dynamic behaviors of the apps. Afterward, the system summarizes and represents captured behaviors in the form of graphs or runtime fingerprints. These are later sent to the Monet server to match against known malware for detection purposes. Monet's client needs changes in both the Android framework in the Linux kernel to properly function, which makes the deployment very hard on a large scale and for simple end-users.

2.5.4 Discussions

Table 2.2 depicts a comparison between *generic* solutions relying on *feature selection* and *detection strategy* criteria. All the *workstation* solutions [63, 91, 148] employ static analysis to extract malicious and benign features because it is fast and allows to detect most malicious apps. These proposals are resilient to common code transformation techniques, depending on the strength of the chosen features [148]. However, these proposals are not immune to more advanced obfuscation techniques such as encryption, especially if the solution uses relatively simple static features[63]. For this reason, Stromdroid [91] leverages dynamic analysis to enhance the detection rate against obfuscated malicious apps. Stormdroid [91] and MAMADROID [148, 158] apply supervised machine learning techniques; in contrast, Opsec [63] relies on fuzzy signatures to identify malicious apps. In the case of *mobile-based* solution, the authors of Drebin [73, 166] efficiently extract simple static analysis features (such as permissions) to minimize the needs in terms of resources; also, they

apply supervised machine learning, classification [73] and anomaly detection [166]. Furthermore, [73] and [166] provide a passive notification to the end-user in contrast to Droidbarrier [68], which prevents malicious actions. To do so, Droidbarrier [68] leverages runtime behaviors in the mobile device to make a detection decision (authentification mechanism). *Hybrid* solutions [81, 85, 181] rely on behavioral features collected from mobile devices to provide passive alarms for the end-users.

Solution	**Feature selection**				**Detection strategy**		
	Type	**Resiliency**	**Adapt**	**Approach**	**Scope**	**Mode**	**Response**
Stormdroid [91]	Hybrid	Highest	Medium	Learning	Known, Zero-day	Supervised	/
Mamadroid [148]	Static	High	High	Learning	Known, Zero-day	Supervised	/
OpSeq [63]	Static	Medium	Low	Signature	Known, Attribution	/	/
Drebin [73]	Static	Medium	Low	Learning	Known, Zero-day	Supervised	Passive
Droidbarrier [68]	Behavioral	Highest	High	Knowledge	/	/	Active
ML [166]	Static	Medium	Medium	Learning	Known	Supervised	Passive
Crowdroid [85]	Behavioral	Highest	High	Learning	Known, Zero-day	Unsupervised	Passive
DRACO [81]	Hybrid	Highest	High	Learning	Known, Zero-day	Supervised	Passive
Monet [181]	Hybrid	Highest	Highest	Signature	Known	/	Passive

Table 2.2: Qualitative Comparison of General Malware Attack Solutions

Table 2.3 depicts quantitative comparison between *general* solutions. *Apk preprocessing complexity* varies between solutions. The highest complexity (StormDroid [91]) is related to the dynamic analysis of an Apk file in a sandbox environment; also, this might require to repackage the Apk file [81] to include the monitoring APIs. A lower complexity [73] is achieved with decompressing Apk file. The lowest Apk processing complexity solution [68] could not use the Apk file because it monitors the running apps that are already installed in the mobile device. *Feature extraction complexity* could be less costly even though the solution uses dynamic analysis because the process of extracting is applied to simple logs files, in addition to the static features, such as StromDroid [91], which requires a relatively small amount of time for the feature extraction. In contrast, MAMADROID [148] uses static analysis, yet its feature extraction is very complex and time-consuming (13-18 minutes per app), which is very high. The *model generation computation* could be very light in case of signature-based solutions [63], [181] or classification-based solutions [91] using *k-nearest neighbor* technique; because the solution needs only to fetch the most similar app in the signature database or the training set. Most solutions show very high detection rates. However, their evaluation dataset size variability drastically affects the generalization of the performance results. For instance, MAMADROID has $88k$ apps and Drebin has 130k apps in their

dataset, while Crowdroid [85] has $0.06k$ apps. The detection rate, time, and dataset size in Table 2.5 are collected from the original publications of the solution.

Solution	Apk Preprocessing Complexity	Feature Extraction Complexity	Model Generation Computation	Detection Rate	Detection Time	Dataset
Stormdroid [91]	Highest	Medium	Lowest	Acc=93.8%	201 s	Ben=4.4k, Mal=3.7k
Mamadroid [148]	High	Highest	Low	F1=99%	13-18 min	Ben=8.5K, Mal=35.5K
OpSeq [63]	Medium	Medium	Lowest	F1=98%	11.6 s	Ben=0.4K, Mal=1.2K
Drebin [73]	Low	Low	Low	Acc=94%	10 sec	Ben=124K, Mal=5.6K
Droidbarrier [68]	Lowest	Lowest	Lowest	/	/	Ben=0K, Mal=1.3K
ML [166]	Meduim	High	Low	F1=85%	/	Ben=2.1K, Mal=0.1K
Crowdroid [85]	Low	Low	High	Acc=85%-100%	/	Ben=0.05K, Mal=0.01K
DRACO [81]	Highest	Low	Low	Acc=98.4%	96 s	Ben=18K, Mal=10k
Monet [181]	High	High	Lowest	Acc=99%	60 s	Ben=0.5K, Mal=3.8K

Table 2.3: Quantitative Comparison of General Malware Attack Solutions

As shown in Table 2.4, depending on the deployment architectures of the general solutions, we could classify them into *workstation*, *mobile*, and *hybrid* based architecture. *Workstation* solutions such as [63, 91] target Android app layer for malware detection. In contrast, Droidbarrier [68] (*mobile-based*), is implemented in the Linux Kernel layer. Furthermore, Monet [181] (*hybrid* solutions) is implemented across two layers, Android framework and app layers.

2.6 Specific Malware Threat Detection

In this section, we present the *attack-based* detection solutions. These solutions still consider the detection of malicious apps but they target a specific malicious behavior in the malicious app

Solution	Layer	Architecture	Physical Components	Target application
Stormdroid [91]	App Layer	Workstation	Server	Malware Detection
Mamadroid [148]	App Layer	Workstation	Server	Malware Detection
OpSeq [63]	App Layer	Workstation	PC	Malware Detection
Drebin [73]	App Layer	Mobile	Mobile/IoT devices	Malware Detection
Droidbarrier [68]	Linux Kernel	Mobile	Mobile/IoT devices	Malware Detection
ML [166]	App layer	Mobile	Mobile devices	Malware Detection
Crowdroid [85]	Framework, App layers	Hybrid	Mobile/IoT devices, Server	Malware Detection
DRACO [81]	App Layer	Hybrid	Mobile/IoT devices, Server	Malware Detection
Monet [181]	App, Framework layers	Hybrid	Mobile devices, Server	Malware Detection

Table 2.4: Classifications of General Malware Attack Solutions

such as *sensitive data leakage*, *GUI-phishing*, and *repackaging*. In addition, we position *attack-based* solutions based on the deployment classification, specifically, into *workstation*, *mobile*, and *hybrid* architecture based solutions.

2.6.1 Workstation-Based Solutions

The following solutions constitute *workstation-based* solutions, in which the authors target large-scale deployment.

ICCDetector [191] is a machine learning system to detect ICC-based (Android Inter-Component Communication) malicious apps. The authors adopt the following approach for the proposed system: (i) They extract ICC features for the Android app using a preexisting tool called EPICC. (ii) They apply feature selections using Correlation-based Feature Selection (CFS). (iii) Finally, the selected ICC features using EPICC are normalized in feature vectors to be input to a binary classifier (SVM, Decision Tree, Random Forest). The feature vector stores the occurrence number of a given feature. In the detection phase, the ICCDetector system uses the trained model to detect ICC-based malware. Using ICC for malware detection, ICCDetector aims to fill the detection of ICC-based malware gap that relies on ICC malicious payload.

AnDarwin [95, 96] detects similar apps that are written by the same developer as well as different developers. AnDarwin consists of four stages: (i) AnDarwin extracts similar vectors by computing an undirected PDG (Program Dependence Graph) of each method in the app using only data dependencies for the edges. (ii) AnDarwin finds similar code segments by clustering all the

vectors of all apps. It identifies code clones by finding near-neighbors of vectors using Locality Sensitive Hashing (LSH). (iii) AnDarwin eliminates library code based on the frequency of the clusters. (iv) AnDarwin detects similar apps (full and partial detection) by computing the pairwise similarity between all the sets using LSH techniques, specifically MinHash. Because AnDarwin is based on PDG for extracting semantics vectors, it is less scalable with respect to analysis time. Using 75 threads, AnDarwin takes 4.47 days to extract semantic vectors (first stage) from 265,359 apps.

FlowDroid [74] performs a context, flow, object, and field-sensitive static taint analysis on Android apps. It models Android app's lifecycle states and handles taint propagation due to callbacks and User Interface (UI) objects. It also utilizes SuSi [161], a machine-learning-based technique, to automatically identify sources and sinks of sensitive information in an Android API. FlowDroid achieves 86% precision and 93% recall, which represent better results than two commercial tools: AppScan and Fortify SCA. However, the system uses high-complexity tools to process Apk files, such as Soot [142] and Dexpler [79]. Also, it does not track data flows across different app components that communicate using Android ICC.

MassVet [88] is designed for vetting apps at a massive scale, without knowing what malware looks like and how it behaves. It runs a DiffCom analysis of the submitted app against the whole market app. It compares a submitted app with all apps already on the market by focusing on the difference between those sharing a similar UI structure, which is known to be primarily preserved during repackaging. It also performs an intersection analysis to compare the new apps against existing apps with different view structures and signed by various certificates. The aim is to inspect their common parts to identify suspicious code segments (at the method level).

2.6.2 Mobile-Based Solutions

In the following, we present *mobile-based* solutions that are meant to be deployed on smart mobile devices.

Aurasium [195] automatically repackages an application to attach a user-level policy enforcement code. The role of this code is to monitor any security violations, such as sending SMS to premium charging numbers. If such a case occurs, Aurasium displays the destination number and the SMS content, so the user can confirm or deny the operation.

AppGuard [76] is a policy-enforcement system, which provides the user with the ability to revoke permissions after app-installation time. It takes an untrusted app and user-defined security policies as input and inserts a security monitor API into the untrusted app by repackaging the Apk. Security policies may specify restrictions on method invocations as well as secrecy requirements. This requires the identification of relevant methods at the API level for which such checks are required.

Patronus [182] is a device-based Intrusion Prevention System (IPS). The latter is a rule (policy) based system, in which there is a policy database that defines what malicious behaviors are. Patronus comprises two main parts: the client side app and the server side service. Both parts are on Android devices. The client is a simple Android app; however, the server-side only updates the policy files in the Android OS. Therefore, there is no need to change the actual Android OS; it only requires to inject Patronus into the system to build a hook and capture app's behaviors. Patronus needs a special privilege to insert such files, which prevents a large scale deployment. Also, injecting files into Android OS triggers some security concerns on these files. Finally, the policy-based detection system is limited to the expressiveness of the rules in the database.

2.6.3 Hybrid Solutions

This section depicts solutions that leverage *hybrid* architecture.

XDroid [160] is proposed as a risk assessment tool and a user alert generator. XDroid has two components: (i) XDroid client, which monitors app's behaviors, such as Android API calls, and timestamps these events. These events are sent continuously to an XDroid Server, where risk assessment, user profiling, and alert customization services leverage time-series of events to assess the suspicious behavior of a given app. To do so, the authors propose the use of Hidden Markov Model (HMM), which is first trained on malicious and benign behavior services. Afterward, the trained HMM Model is deployed to server. However, XDroid needs a special privilege in Android devices to be able to log app's behaviors; this could prevent this technique from having large-scale deployments since devices need to be rooted to track app behaviors properly.

DroidEagle [180] uses the layout resources within an app to detect visually similar apps, a common characteristic in repackaged apps and phishing malware. To discover visually similar apps,

DroidEagle consists of two sub-systems: RepoEagle and HostEagle. RepoEagle performs large-scale detection on apps repositories (e.g., apps markets), and HostEagle is a lightweight mobile app that can help users to detect visually similar Android apps quickly upon download. The reported performance results show that, within 3 hours, RepoEagle can detect 1298 visually similar apps from 99,626 apps in a repository.

2.6.4 Discussions

Table 2.5 shows the comparison between *attack-based* solutions using *feature selection* and *detection strategy* criteria. As shown in Table 2.5, most *Workstation-based* solutions are based on static analysis. In addition to code static analysis, some solutions statically analyzed the resources of the Apk file, for instance, MassVet [88] leverages only the GUI XML resources files in the Apk and ignore the rest of the content. Furthermore, *Workstation-based* solutions tend to be resilient against known obfuscation techniques since such technique mainly target bytecode, while these solutions do not rely only on bytecode to detect Android malicious apps. On the other hand, *mobile-based* solutions [76, 182, 195] rely only on behavioral features of app runtime on the end-users mobile devices. These features are highly resilient to obfuscation and adapt to OS changes since they are collected from executions of the malicious apps. Both Aurasium [195] and AppGuard [76] provide a passive notification to end-users; Patronus [182] goes a step further by blocking the detected malware. Also, *mobile-based* solutions rely on policy rules to prevent malicious behaviors. *Hybrid* solutions employ behavioral analysis as in XDroid [160] and static analysis as proposed in Droideagle [180]. The latter is resilient to code obfuscation because it relies on GUI similarity and does not consider the bytecode.

Table 2.6 shows a quantitative comparison between *attack-based* solutions. *Apk pre-processing complexity* varies among the solutions; it is medium in some solutions [74, 88] and high in other solutions [96, 191] that use static analysis because the pre-processing of Apks. In case of Aurasium [195] and AppGuard [76] the *Apk pre-processing complexity* is high because of the Apk's repackaging complexity to inject the monitoring Apk hooks. It is low for XDroid [160] and Droideagle [180] because the first [160] relies mainly on runtime traces and the second [180] uses Apk GUI resources for its graphical signatures. Finally, *attack-based* solutions have very low complexity for

Solution	Feature selection			Detection strategy			
	Type	Resiliency	Adapt	Approach	Scope	Mode	Response
ICCDetector [191]	Static	High	Low	Learning	Known	Supervised	/
Andarwin [96]	Static	High	High	Signature	Known	/	/
Flowdroid [74]	Static	High	High	Prior-knowledge	Known, Zero-day	/	/
MassVet [88]	Static	High	High	Signature	Known, Zero-day	/	/
Aurasium [195]	Behavioral	Highest	Highest	Prior-knowledge	Known, Zero-day	/	Passive
AppGuard [76]	Behavioral	Highest	Highest	Prior-knowledge	Known, Zero-day	/	Passive
Patronus [182]	Behavioral	Highest	Highest	Prior-knowledge	Known, Zero-day	/	Active
XDroid [160]	Behavioral	Highest	Highest	Learning	Known	Unsupervised	Active
Droideagle [180]	Static	High	High	Signature	Known, Zero-day	/	Passive

Table 2.5: Qualitative Comparison of Specific Malware Attack Solutions

model generation because most of these models are based on policy rules databases.

Solution	APK Preprocessing Complexity	Feature Extraction Complexity	Model Generation Computation	Detection Rate	Detection Time	Dataset
ICCDetector [191]	High	Low	Low	Acc=97.4%	about 40 s	Ben=12k, Mal=5.3k
Andarwin [96]	High	High	Lowest	Found=36k Rebranded App	10h	Apps=266k
Flowdroid [74]	Medium	High	Lowest	F1=90%	16 s	Ben=1k, Mal=0.5k
MassVet [88]	Medium	Medium	Lowest	Acc=72%	10 s	Apps=1.2 Million
Aurasium [195]	High	Lowest	Lowest	Seccess=99.6%	/	Ben=3.5k, Mal=1.3k
AppGuard [76]	High	lowest	Lowest	/	about 20 s	Apps=250k
Patronus [182]	Low	Lowest	Lowest	F1=69-92%	/	Ben=0,5k, Mal=0.3k
XDroid [160]	Low	Low	Low	F1=83%	/	Benn=0.7k, Mal=5.6k
Droideagle [180]	Low	Low	Lowest	Similar=1.3K apps	Average=62 s	Apps=100k

Table 2.6: Quantitative Comparison of Specific Malware Attack Solutions

Similarly to *generic* class, the *attack-based* solutions target Android malware detection. However, these solutions target specific malicious attacks, in contrast to *general solutions*, which target malware in general. For instance, ICCDetector [191] aims to detect malware that employ Android Inter-Components Communication (ICC) to their attack. Another example, Flowdroid [74] helps detecting sensitive data leakages, which is a known pattern of malicious apps. Table 2.4 shows that *workstation-based* solutions [191] [74, 88, 96] rely on app layers of Android stack; the other solutions (*mobile-based* and *hybrid*) are implemented across app and framework layers. The exception here is AppGuard [76], which is implemented on Linux kernel layer.

Solution	Layer	Architecture	Physical Components	Target application
ICCDetector [191]	App Layer	Workstation	PC	ICC Abuse
Andarwin [96]	App Layer	Workstation	Server	Find Repackaging
Flowdroid [74]	App Layer	Workstation	PC	Sensitive Data Leakage
MassVet [88]	App Layer	Workstation	PC	Find GUI Repackaging
Aurasium [195]	App, Framework Layers	Mobile	Mobile/IoT devices	Policy Enforcement
AppGuard [76]	Linux Kernel	Mobile	Mobile/IoT devices	Policy Enforcement
Patronus [182]	App, Framework Layers	Mobile	Mobile devices	Mobile IPS
XDroid [160]	App, Framework layers	Hybrid	Mobile/IoT devices, Server	Risk Assessment
Droideagle [180]	App, Framework layers	Hybrid	Mobile devices, Server	GUI Phishing and Repackaging

Table 2.7: Classifications of Specific Malware Attack Solutions

2.7 Android Malware Detection Helpers

In this section, we present systems that help enhance Android malware detection systems. These solutions do not provide a malware detection system, but they are used as tools to enhance the malware detection in term of accuracy and runtime performance. The following solutions are lab tools that could be leveraged for malware detection.

The authors in [75] analyze the Android framework statically. The authors propose a top-down approach in their analysis by taking the source code of the Android framework as input. For this purpose, the authors face different challenges: (i) The Android framework layer is different from the application layer, so the existing analysis tools and techniques can not be used to analyze the framework layer. (ii) The framework services may be queried simultaneously from multiple apps. Thus, the framework layer uses various multi-threading mechanisms. (iii) It is unclear what resources are protected by Android permissions. The authors use their analysis insights to develop the Axplorer tool [75] for the analysis of the Android framework layer. To demonstrate the effectiveness of their security analysis, the authors conduct a permission API mapping (map a given permisson to the related Android Framework API). In comparison to previous research, they achieve a more precise analysis. Finally, they propose a permission locality security concept to measure permission coverage overlap of Android APIs.

DroidChameleon [162] is a tool to evaluate the robustness and resiliency of anti-malware solutions against the state-of-the-art obfuscation techniques. DroidChameleon provides a set of obfuscation techniques to be tested on targeted malware detection systems. DroidChameleon supports three levels of obfuscation techniques: (i) Trivial obfuscations, that do not change the bytecode but

only repackages API file or disassembling and Reassembling the Dex file. (ii) Attacks detectable by static analysis obfuscations that make a change to the bytecode in one or multiple ways such as identifier renaming, code reordering, junk code insertion, which could be detected using static analysis. (iii) Attacks that could not be identified using static analysis such as reflection and bytecode encryption. The authors conduct a systematic evaluation of existing anti-malware products against various obfuscation techniques; this is a necessary evaluation to measure the resiliency of existing malware detection systems.

2.7.1 Discussions

Demystifying [75] proposes an in-depth analysis of the Android framework. As an application for this analysis, they reevaluate Android permission mapping to the actual Android assets with high precision. Having such precise mapping helps in malware detection. Given an Android app the detection system could map permissions to a more granular view by using Android assets. On the other hand, the authors of DroidChameleon [162] propose a tool that provides a set of obfuscation techniques to be applied to Android apps. DroidChameleon is a valuable tool that helps enhancing malware detection system by evaluating these systems on obfuscated malware and benign sample using DroidChameleon. In a security lab environment, one could ensure that the detection system is resilient to DroidChameleon obfuscation techniques. A simple classification of Android malware detection helpers is depicted in Table 2.8. In this thesis, we use DroidChameleon obfuscation tool to build an Android obfuscation dataset that will be used for the evaluation of elaborated systems in the next chapters.

Solution	**Layer**	**Architecture**	**Physical Components**	**Target application**
Demystifying [75]	Framework Layer	Workstation	Server	Android Framework Analytics tool
DroidChameleon [162]	App Layer	Workstation	PC	Obfuscation Tool

Table 2.8: Classification of Android Malware Detection Helpers

2.8 Summary

In this chapter, we reviewed selected prominent Android malware detection solutions and proposed a taxonomy to classify them. This taxonomy allows to carry out an in-depth comparative study between the proposals of the same category. By proposing a generic functional framework for Android malware detection, we defined the performance criteria used for our comparative study. Through the study of different Android malware detection systems, we define the advantages and disadvantages of each system approach. Moreover, we illustrated the situations where each approach exhibits good performance.

Android detection systems with high detection accuracy incur a long processing time for complex feature extraction, which implies a high detection latency and the use of more computation resources. On the other hand, detection systems with less sophisticated feature extraction achieve a moderate detection accuracy with the advantage of low detection latency and small resources needed, which allows such systems to function properly on mobile and IoT devices. To this end, an appealing future work lies in achieving simultaneously high detection results and a low detection latency with the minimum computation resources to fit all scales of devices.

In the next chapter, we propose a solution for Android malware clustering using static analysis features and graph partitioning algorithms. According to the proposed taxonomy discussed in this chapter, we position this solution as (i) general-attack oriented because it targets the detection of all types of Android malware. Also, we consider it as (ii) workstation oriented due to the level of resources it requires in the deployment. Finally, it is an application layer solution because it considers only Android apps.

Chapter 3

Robust Android Malicious Community Fingerprinting

3.1 Overview

Security practitioners can combat large-scale Android malware by decreasing the *analysis window* size of newly detected malware. The window starts from the first detection until signature generation by anti-malware vendors. The larger the window is, the more time malicious apps are given to spread over the users' devices. Current state-of-the-art techniques have a large analysis window due to the significant number of Android malware appearing daily. Besides, these techniques use manual analysis in some cases to investigate malware. Therefore, decreasing the need for manual detection could significantly reduce the *analysis window*. To address the aforementioned issue, we elaborate systematic techniques and tools for the detection of both known family apps and new malware family apps (i.e., variants of existing families or unseen malware). To do so, we rely on the assumption that any pair of Android apps, with distinct authors and certificates, are most likely to be malicious if they are highly similar. Because the adversary usually repackages multiple app packages with the same malicious payload to hide it from anti-malware and vetting systems. Consequently, it is difficult to detect such malicious payloads from benign functionalities of a given Android package. Accordingly, a pair of Android apps should not be very similar in their components, excluding popular libraries. This observation, as mentioned earlier, could be used to design

and develop a security framework to detect Android malware apps.

In this chapter, we propose a novel Android app fingerprinting technique, APK-DNA, inspired by fuzzy hashing. We specifically target fingerprinting Android malicious apps. Computing the APK-DNA of a suspicious app requires a low computation time. Afterward, we leverage the previously-mentioned assumption (i.e., very similar apps might be malware from the same malware family) to propose a cyber-security framework, namely Cypider (*Cyber-Spider For Android Malware Detection*), to detect and cluster Android malware without prior knowledge of Android malware apps. Cypider consists of a novel combination of a set of techniques to address the problem of Android malware, clustering, and fingerprinting. First, Cypider can detect repackaged malware (malware families), which constitute the vast majority of Android malware apps [203]. Second, it can detect new malware apps, and more importantly, Cypider performs the detection automatically and in an unsupervised way (i.e., no prior knowledge about the apps). The fundamental idea of Cypider relies on building a *similarity network* between the targeted apps static content in terms of fuzzy fingerprints. Actually, Cypider extracts, from this *similarity network*, sub-graphs with high connectivity, called *communities*, which are most likely to be *malicious communities*.

3.1.1 Threat Model

In the context of this chapter, the focus is on detecting malware targeting Android mobile apps with no prior knowledge about the malware. In particular, instead of focusing on the detection of an individual instance of malware, Cypider targets bulk detection of malware families and variants as malicious communities in the similarity network of the apps dataset. Moreover, Cypider aims for a scalable yet accurate solution that can handle the overwhelming volume of the daily detected malware, which could aggressively exploit users' smart devices. Cypider is robust (Section 3.6) but not immune against obfuscated apps contents. Cypider could handle some types of obfuscations because it considers different static contents of the Android package in the analysis. This makes Cypider more resilient to obfuscation as it can group malware apps by also considering other static contents that are not obfuscated, such as app permissions or Android API calls.

3.2 Malware Fingerprints

A fuzzy (hashing) or approximate fingerprint of binary software is a digest that captures its static content, in similar manner to cryptographic hashing fingerprints such as MD5 and SHA1. Still, the fuzzy fingerprint change is virtually linear to the change in the binary content. In other words, smaller changes in the static content of the malware will cause a minor change in the computed fuzzy fingerprint. In the context of cybersecurity, this is an important property that helps in detecting polymorphic malware attacks. Current fuzzy fingerprints such as *ssdeep*[1] are computed for the app binary as a whole, which makes them less effective for detecting malicious app variations. This problem gets complicated in the case of Android OS due to the apps packaging structure, which contains not only the actual compiled code but also other content such as media files. To overcome this limitation, we propose an effective and broad fuzzy fingerprint that captures not only binary features but also the underneath structure and semantics of the *APK* package.

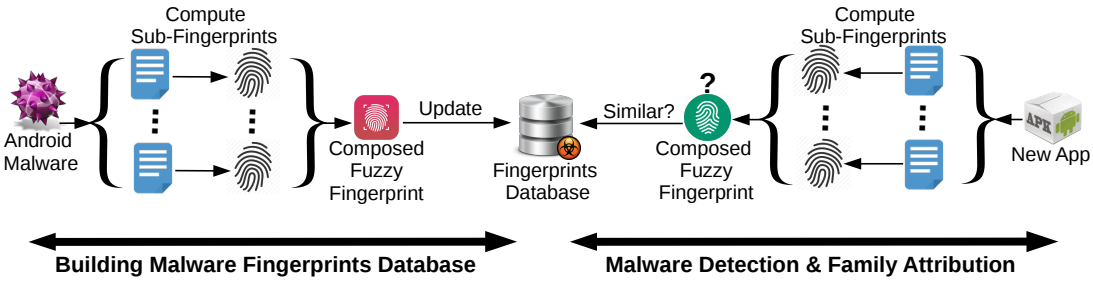


Figure 3.1: Approximate Fingerprint Approach

Accordingly, our approach for computing Android app fingerprints relies on decomposing the actual *APK* file into different content categories. For each category, we compute a customized fuzzy hash (sub-fingerprint). Note that for some categories, for instance, *Dex* file, the application of the customized fuzzy hashing on the whole category content does not capture the structure of the underlying category. In this case, we apply fuzzy hashing against a selected *N-grams* of the category content. In our context, we use *byte n-grams* on binary files and *instruction n-grams* on assembly files. Furthermore, a best practice in malware fingerprinting is to increase the entropy of the app package content [151]. To this end, we compress each category, as proposed in [151]

[1] https://ssdeep-project.github.io/ssdeep/index.html

to increase the entropy, content before computing the customized fuzzy hash on its N-grams. The resulting fuzzy hashes (sub-fingerprints) are then concatenated to produce the final composed fuzzy fingerprint, called APK-DNA. As depicted in Figure 6.1, there are two main processes: First, we build a database of fingerprints by generating APK-DNAs of known malware samples to identify whether a new app is malicious or not and to attribute the family of malicious ones. Second, for each malware under investigation, we compute its APK-DNA and match it against existing fingerprints in the database of known malware samples.

By generating fuzzy fingerprints for all known malware (in this scenario, we need fingerprints of known malware to make the detection; however, as will be presented in Section 3.3, Cypider employs APK-DNA for a completely unsupervised approach, i.e., no prior knowledge on apps is required), the system is ready for detection and family attribution. Thus, the detection process starts by computing APK-DNA fingerprints for known Android malware. We use multiple compression schemas for testing purposes. Thus, in the final fingerprint of the *APK*, only one compression is used. Moreover, we use APK-DNA fingerprints as a basis to design and implement ROAR, a novel framework for malware detection and family attribution. ROAR's first approach, namely *family-fingerprinting*, computes a fingerprint for each malware family. Afterward, it uses these family fingerprints to make security detection decisions on new apps. In the second approach, *peer-matching*, ROAR uses the whole fingerprint database for detection and attribution.

3.2.1 Approximate Static Fingerprint

In this section, we present our approach for Android apps fingerprints generation.

Fingerprint Structure

We leverage the aforementioned *APK* structure to define the most important components for fingerprinting. The design of the *APK* fingerprint must consider most of its important components as unique features to distinguish between different malware samples. As depicted in Figure 3.2, APK-DNA is composed of three main sub-fingerprints based on their content type: **Metadata**, **Binary**, and **Assembly**. The **Metadata** sub-fingerprint contains information, which is extracted from the **AndroidManifest.xml** file. We particularly focus on the required permissions. The aim is to

fingerprint the permission used for a specific Android malware or malware family. Our intuition stems from the fact that some Android malware samples need specific types of permissions to conduct their malicious actions. For example, the **DroidKungFu1** malware [203] requires access to personal data to steal sensitive information. Having Android malware without access permissions to personal data, e.g., *phone number*, would suggest that this malware is most likely not part of **DroidKungFu1** family. Other metadata information could be considered for malware segregation, for instance, *Activity* list, *Service* list, and *Component*. In the current design of APK-DNA, we focus on the required access permissions.

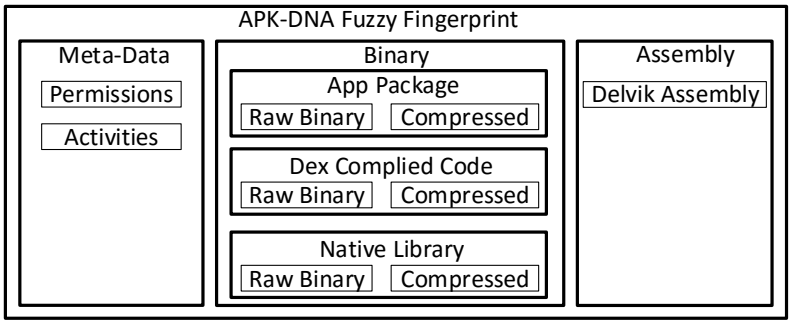


Figure 3.2: Android Package Fingerprint Structure

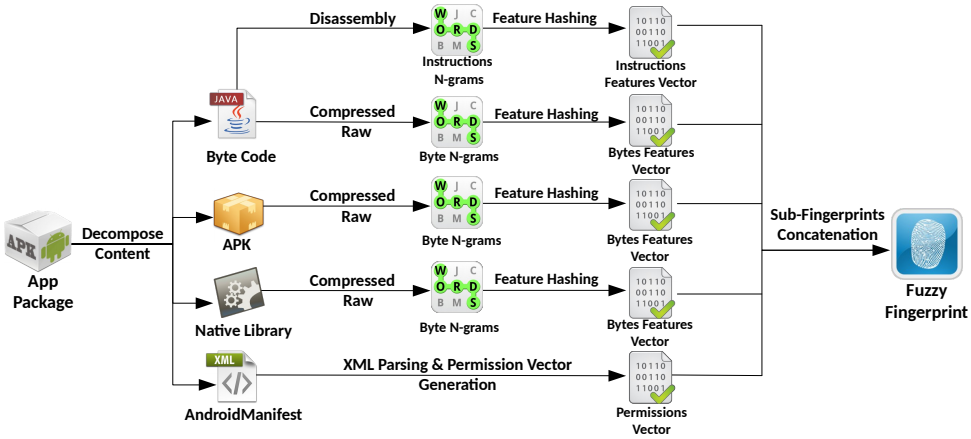


Figure 3.3: Fingerprints Computation Process

The **Binary** sub-fingerprint captures the binary representation of the *APK* file content. In other words, we aim to fingerprint the byte sequence of Android malware. In this context, we use *n-grams* [151] as we will present in Section 3.2.1. We divide the binary sub-fingerprint into three parts: **App**

Package, **Dex Compiled Code**, and **Native Library**. The **App Package** consists of the *APK* file. Thus, all the components inside the package are considered (e.g., media file). Along with the raw *APK* package, we apply a compression schema on the package to increase its *entropy* [151]. In the **Dex Compiled Code**, we focus on the code section of the Android malware, which is located in the *Dex* file of Android apps. The use of the code section for malware detection has proven its accuracy [122]. In the context of Android malware, we use extracted features from the *classes.dex* as part of the APK-DNA. Besides, by applying compression, we use a high-entropy version of the *classes.dex* for fingerprinting. The **Native Library** part of the binary sub-fingerprint captures C/C++ shared libraries, used by malware. Using the native library for malware fingerprinting is essential in some cases, for example, to distinguish between two Android malware samples. For instance, if the malware uses a native library, it is more likely to be **DroidKungFu2** rather than **DroidKungFu1** because **DroidKungFu2** malware family uses C/C++ library and **DroidKungFu1** uses only *Java bytecode*.

In the **Assembly** sub-fingerprint, we also focus on the code section of Android malware, which is *classes.dex*. However, we do not consider the binary format. Instead, we use the reverse-engineer assembly code. As we will present in Section 3.3.3, we reverse-engineered the *Dalvik byte-code* in order to extract instruction sequences used in the app. The **Assembly** sub-fingerprint aims to discriminate malware using the unique instruction sequences in the assembly file. We use the same technique as in the **Binary** sub-fingerprint, i.e., *n-grams*. However, here we consider the assembly instructions instead of bytes. In addition to assembly instructions, we could also consider *section names*, *call graphs*, etc. In the current design, we focus on the assembly instructions for fingerprinting.

Fingerprints Generation

In this section, we present the steps required to generate APK-DNA fingerprints. In addition, we present the main techniques adopted in the design of the fingerprint, namely, *N-gram* and *Feature Hashing*. Afterward, we show the similarity techniques that are employed to compare *APK* fingerprints.

N-grams. The *N-gram* technique is used for computing contiguous sequences of *N* items from a large sequence. For our purpose, we use N-grams to extract the sequences (the order is important within the n-gram sequence) used by Android malware to be able to distinguish between different malware samples. To increase the fingerprint accuracy, we leverage two types of N-grams, namely *instruction N-grams* and *bytes N-grams*. As depicted in Figure 3.4, the instruction N-grams are the unique sequences in the disassembly of a *Dex* file, where instructions are stripped from the parameters. In addition to instruction N-grams, we also use byte N-grams on different contents of the Android package. Figure 3.4 illustrates different N-grams on both instructions and bytes of the first portion of the **AnserverBot** malware. We have experimented with multiple options such as *bigrams*, *3-grams*, and *5-grams*. The last one provided the best results in the design of APK-DNA fingerprint, as will be shown in the evaluation section. The result of N-grams extraction is the list of unique *5-grams* for each content category, i.e., *assembly instructions*, *classes.dex*, *native library*, and *APK file*.

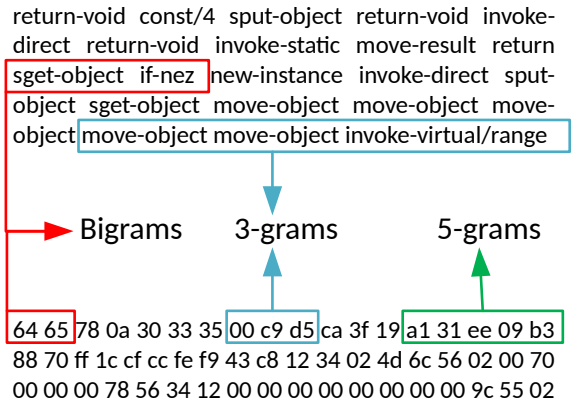


Figure 3.4: Instructions and Bytes from AnserverBot Malware

Feature Hashing *Feature hashing.* is a machine learning preprocessing technique for squashing an arbitrary number of features into a fixed-size feature vector. The feature hashing algorithm, described in Algorithm 1, takes as input the set of sequences generated by applying the N-gram technique and the length of the output feature vector. In the current implementation of APK-DNA, we use a bit feature vector of 16KB. However, the size could be adjusted according to the needed density of the bit-feature vector to distinguish between apps. For example, the size of the assembly

instruction vector could be less than the dex vector since the density produced by the instruction content is less than the dex one. Notice that in our implementation, we store only a binary value, which defines whether the N-gram exists or not. The standard feature hashing uses the frequency, i.e., the number of occurrences of a given N-gram. The output of the feature hashing algorithm is a feature-bit vector. Instead of using existing fuzzy hashing algorithms such as *ssdeep*, we leverage the feature vector as our fuzzy hashing technique for implementing APK-DNA fingerprint. In the next section, we present the complete process of computing the fingerprint using N-grams and feature hashing as basic blocks.

Algorithm 1: Feature Vector Computation

```
input  : N-grams: Set,
         L: Feature Vector Length
output: Binary Feature Vector
features_vector = new bitvector[L];
for Item in N-grams do
    H = hash(Item) ;
    feature_index = H mod L ;
    features_vector[feature_index] = 1 ;
end
```

Fingerprint Computation Process. As shown in Figure 3.3, the fingerprint computation process starts by decomposing the Android app *APK* file into four different content categories: 1) *Dalvik byte-code*, 2) *APK file*, 3) *native libraries*, and 4) *AndroidManifest file*. Each binary content is compressed to increase the entropy. Afterward, we extract the byte N-grams from the raw assembly and the compressed content. The resulting set of N-grams is provided as input to the feature hashing function to produce the customized fuzzy hashing. The size of each customized fuzzy hash is 16KB, as mentioned in Section 3.2.1. For *Dalvik bytecode*, we fingerprint the assembly code in addition to the binary fingerprint. First, we reverse engineer the *Dex* file to produce its assembly code. After preprocessing the assembly, we use the instruction sequence of the Android app to extract the instruction N-grams set. Afterward, we use feature hashing to generate a 16KB bit vector fingerprint for the assembly code. The current design of APK-DNA uses the *feature hashing* technique without

feature selection because we aim to keep maximum information on the targeted malware instance or its family. However, *feature selection* could be a promising technique to explore in future APK-DNA design.

Regarding the *AndroidManifest file*, we first convert it into a readable format, then parse it to extract the required permissions by the Android app. To use the required permission app fingerprinting, we use a bit vector of all Android permissions in a predefined order. For each given required permission, we flag the bit to 1 in the permission vector if it exists in the *AndroidManifest file*. The result is a bit vector for all the permissions of the Android app. At the end of the operations mentioned above, we generate five-bit vectors. The final step of the fuzzy fingerprint computation consists of concatenating all the produced digests into one fingerprint, designated as APK-DNA. It is important to mention that, for similarity computation, we also keep track of the bits of each content vector. Notice that the content categories are mandatory for Android apps except the *native library*, which may not be part of the app. Therefore, we use a bit vector of zeros for the feature vector of the *native library*. The final size of APK-DNA is 16KB for the feature vector of each content (there are four feature vectors: assembly, bytecode, APK, and native library). However, for the permission vectors, we use a 256-bit feature vector since the Android permission system does not exceed this number.

Compute Fingerprints Similarity. The main reason for adopting the feature vector as a customized fuzzy hash is to make the similarity computation straightforward using *Jaccard Similarity*, as shown in Equation 1. Since we have a set of bit feature vectors flagging the existence of a feature, we adopt a *bitwise jaccard* similarity, as depicted in Equation 2. The *Jaccard Similarity* is computed by dividing the cardinality of the intersection set by the cardinality of the union set.

$$Jaccard(X,Y) = \frac{|X \bigcap Y|}{|X \bigcup Y|}$$
$$0 \leq Jaccard(X,Y) \leq 1 \tag{1}$$

$$Jaccard\_bitwise(A,B) = \frac{Ones(A \wedge B)}{Ones(A \vee B)}$$
$$0 \leq Jaccard\_bitwise(X,Y) \leq 1 \tag{2}$$

Let A and B be two bit-feature vectors; the union of the two vectors is given by the logical expression $A \vee B$, and its cardinality is the number of "1" bits in the resulting vector. Similarly, the cardinality of the intersection of the two vectors is the number of "1" bits in $A \wedge B$ bit vector. As presented in Section 3.2.1, APK-DNA fuzzy fingerprint is composed of five fuzzy hashes, which are bit-feature vectors. To compute the similarity between two fingerprints, we calculate the bit-wise Jaccard similarity between the bit feature vectors representing the same content. In other words, we calculate the similarity between the feature vectors of the assembly, byte-code, APK, native library, and permissions. The result is a set of five similarity values.

3.2.2 Malware Detection Framework

In this section, we leverage the proposed APK-DNA fingerprint for Android malware detection. More precisely, we present i) the*family-fingerprinting* approach, where we define and use a family fingerprint, and ii) the *peers-matching* approach, where we compute the similarity between malware fingerprints. Both approaches are based on the *peer-fingerprint-voting* mechanism to decide on malware detection and family attribution.

Peer Fingerprint Voting

As we have seen in Section 3.2.1, comparing two Android malware packages consists of computing similarities between their metadata, binary, and assembly sub-fingerprints, which gives numerical values on how the two packages are similar in a specific content category, as presented in Algorithm 2. In addition, we add the summation of all the similarities as a summary value of these sub-contents similarities. Note that other summary values, such as the average and the maximum, could also be used. However, it is challenging to detect the most similar packages if we compare an unknown package to known malware packages using multiple sub-fingerprints. The most obvious solution is to merge bit-vectors of each content category into one vector and then compute the similarity of the resulting feature vector. However, in our case, merging bit vectors will heavily reduce the contribution of some sub-fingerprints in the similarity computation.

Likewise, the density of the assembly feature vector is considerably less compared to the binary feature vector. Consequently, we propose to use a composed similarity using *peer-fingerprint voting*.

Algorithm 2: APK-DNA Similarity Computation

```
input  : APK-DNA A: list
         APK-DNA B: list
output: similarity-list: list
similarity-list = empty-list();
for content in content categories do
    similarity = Jaccard_bitwise(A[content],B[content]) ;
    similarity-list.add(similarity);
end
summation = sum(similarity-list);
similarity-list.add(summation);
```

Algorithm 3: Peer-Fingerprint Voting Mechanism

```
input  : similarity-list A-B: list
         similarity-list A-C: list
output: Decision
A-B-count = 0 ;
A-C-count = 0 ;
for content in content categories do
    if A-B[content] >A-C[content] then
        A-B-count += 1;
    else
        A-C-count += 1;
    end
end
if A-B-count >A-C-count then
    Decision = A-B;
else
    Decision = A-C;
end
```

The idea is to compare parts (sub-fingerprints) instead of comparing full fingerprints, as depicted in Algorithm 3. In other words, we examine each sub-similarity pairs. The decision is made by a voting mechanism on the result of each sub-comparison. Moreover, in case of equal votes, we compare the *summation* of the sub-similarities to remove the ambiguity as shown in the example depicted in Figure 3.5. At this stage, we can compare different Android packages and decide on the most similar package to a given one. In what follows, we propose two approaches to malware detection.

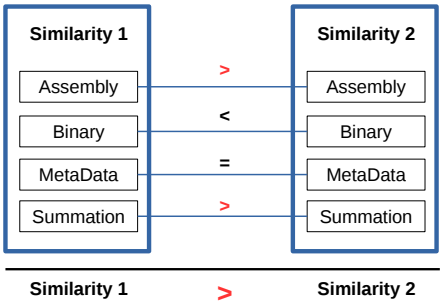


Figure 3.5: Peer-Fingerprint Voting

Peer Matching

In the *peer-matching* approach, ROAR queries the fingerprints database to check the most similar malware fingerprint. To detect Android malware variation, we build a *malware fingerprint* database by computing APK-DNA for known Android malware. The more fuzzy fingerprints in this database, the broader the detection system could cover. As shown in Figure 3.6, for each new malware, we compute its APK-DNA and add it to the database.

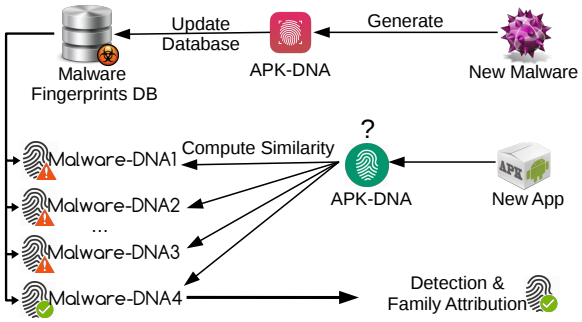


Figure 3.6: Malware Detection Using Peer-Matching

To attribute the malware family to a new app, we first compute the similarity between the malware fingerprint and each entry in the database of known malware fingerprints, as depicted in Figure 3.6. To this end, we use *bitwise Jaccard* similarity, presented in Section 3.2.1, to produce a set of sub-similarity values, i.e., the *composed similarity*. Afterwards, to compare the *composed similarity* values, we use the previously presented *peer-voting* technique. The entry with the highest similarity value that exceeds an acceptance threshold determines the malware family. In the current implementation, we use an experimentally derived static threshold. As such, *Peer-matching* is a simple approach for malware detection and family attribution.

Family-Fingerprinting

In this approach, some extra steps are needed to build a second database of malware family fingerprints. The aim is to reduce the number of database entries required to match an Android malware fingerprint. For this reason, we propose a custom approximate fingerprint for a malware family. The intent is to leverage this family fingerprint for malware detection purposes. The idea is to build a database of family fingerprints from known Android malware samples, and use this database for similarity computation with unknown malware apps. The number of malware families limits the actual size of the family-fingerprints database. Notice that the fingerprint structure for a malware family is the same as for a single malware, i.e., metadata, binary, and assembly family sub-fingerprints.

Algorithm 4: Family Fingerprint Computation

```
input  : Malware Family X Fingerprints: Set
output: Family X Fingerprint: FP_X
FP_X = new bitvector[Zeros];
for fprint in Fingerprints do
    FP_X{meta} = FP_X{meta} or fprint{meta};
    FP_X{bin} = FP_X{bin} or fprint{bin};
    FP_X{asm} = FP_X{asm} or fprint{asm};
end
```

Algorithm 4 depicts the computation of the family fingerprint based on the underlying content sub-fingerprints. First, the fingerprint is initialized to zeros (each content sub-fingerprint). Afterward, the fingerprint is generated by applying a logical *OR* on the current value of the family fingerprint with a single malware fingerprint. Note that each content sub-fingerprint is computed separately. This operation is applied to all malware samples in the database. After calculating the fingerprints from known malware samples, we store them in a *family-fingerprint* database, which is used for detection and family attribution. The detection process is composed of several steps. First, for a given Android package, we generate its fingerprint as described in Section 3.2.1. Then, we compute the similarity between this fingerprint and each family fingerprint in the database. The family with the highest similarity score will be chosen as the family of the new app if the similarity value is above a defined threshold. In the current implementation, we use an experimentally derived static threshold, which is only applied to the *summation* part of the composed similarity. The result

is similar to the single malware fingerprint, but it represents a malware family instead of a particular malware.

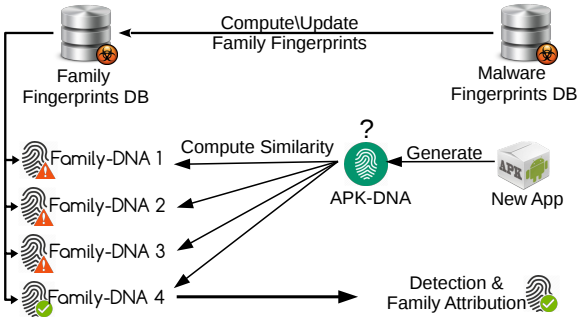


Figure 3.7: Malware Detection Using Family-Fingerprint

3.2.3 Experimental Results

In this section, we present the results in terms of accuracy for both approaches that are adopted in ROAR framework, namely family-fingerprinting and peer-matching.

Testing Setup

Our dataset contains 928 malware samples from Android Malware Genome Project [17, 203]. For evaluation, we selected malware families with many samples since some malware families in Android Malware Genome Project [17] contain only a few samples (some families have only one sample), as depicted in Table 3.1. Clearly, by filtering out other families that do not have enough malware samples, we may miss the detection of these malware families. In addition to known malware samples, we use benign Android applications in each evaluation. These apps have been downloaded from Google play randomly without considering the popularity of the app, as shown in Table 3.1.

For each evaluation benchmark and from the balanced dataset, we randomly sample 70% (70 for training and 30% for testing is a common split method in machine learning) of each family from the dataset to build the fingerprints database. The rest of the dataset (30%) is used for the evaluation of ROAR approaches and sub-fingerprints. Notice that the random sampling is done for every benchmark evaluation. Accordingly, we repeat the assessment five times. The final result is

#	Malware Family/Apps	Number of Samples
0	AnserverBot	187
1	KMin	52
2	DroidKungFu4	96
3	GoldDream	47
4	Geinimi	69
5	BaseBridge	122
6	DroidDreamLight	46
7	DroidKungFu3	309
8	Benign Apps	100

Table 3.1: Evaluation Malware Dataset

the average of the evaluation results.

Evaluation Results

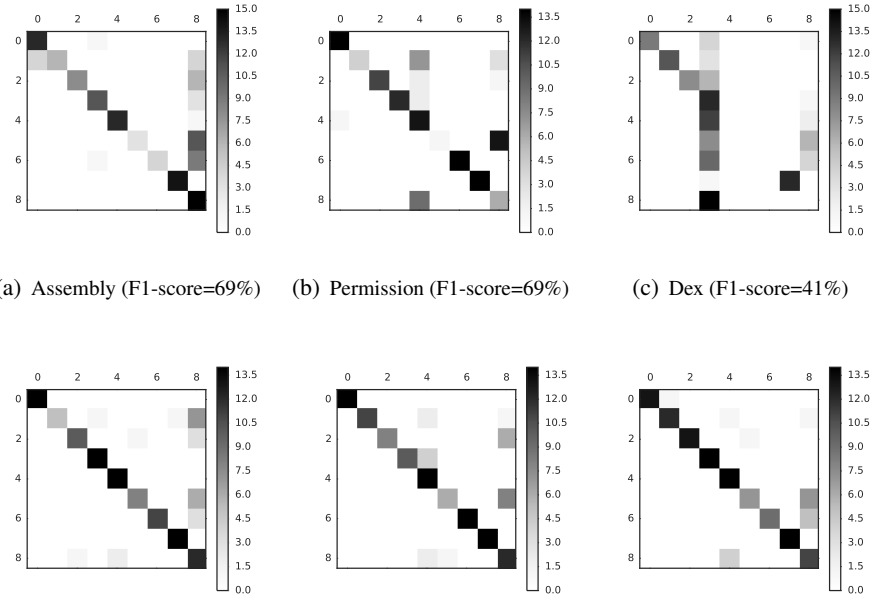
In this section, we present the evaluation results of the ROAR framework. Each approach is separately evaluated. The results are presented using *F1-score*, *precision* and *recall*. The approach is evaluated multiple times (five) using different fingerprint setups, i.e., combinations of sub-fingerprints, which are used to compute the similarities using *peer-voting* technique. Furthermore, we present a comparison between the proposed *peer-voting* similarity technique and the merged fingerprint similarity.

Confusion Matrix Description. In addition to the previous evaluation metrics, we also use the confusion matrices in each evaluation, as shown in Figures 3.8 and 3.9. Each confusion matrix is a square table, where the number of rows and columns are respectively malware families and benign apps following the same order as in Table3.1. The columns and rows from 0 to 7 are respectively the malware families, **AnserverBot, KMin, DroidKungFu4, GoldDream, Geinimi, BaseBridge, DroidDreamLight**, and **DroidKungFu3**, and the column and row 8 represent the benign apps. The interpretation of the confusion matrix results is related to the intensity of the color in its *diagonal*. The darker the color is in the *diagonal*, the higher and the more accurate are the results of the evaluation (*true positive*). The color intensity of the confusion matrix cells represents the number of malware/apps that have been assigned to this cell. However, the less intense is the color in the *diagonal*, and the more intense in the other cells, the less accurate is the result.

False Negative. For any row from 0 to 7 (i.e., malware family), there is a missing malware family attribution if we have a gray color in the other cells of the same row. Even though we missed

in the family attribution task, we still detect the app as malicious. However, a gray cell in column 8 (benign apps) means that we missed bothdetection and family attribution (*detection false negative*). **False Positive**. In the benign apps row (row number 8), the gray color in malware cells indicates that there is a *false positive*. In other words, there are benign apps that are detected as malicious. The number of *false positives* could be measured using the intensity of the color according to the color bar.

Family-Fingerprinting Results. As depicted in Table 3.2, the F1-score, precision, and recall of the *family-fingerprinting* vary according to the fingerprint setup. We evaluated the approach for each content type separately, i.e., *assembly*, *permission*, and *dex* files, so that we can clearly see the impact of each component in the final fingerprint. Both *assembly* and *permission* types show more accurate results compared to *dex* type. Specifically, the *permission* shows a promising result (82% precision), as illustrated in Table 3.2. This indicates the impact of the metadata on Android malware detection. It is that investigating other metadata could result in higher accuracy. The *APK* fingerprint results are surprising because of the poor accuracy value, under 40% f1-score. The learned lesson is that applying the fuzzy fingerprinting (including *ssdeep*) to the whole package could mislead the malware investigation when using fuzzy matching. The confusion matrix for each setup demonstrates a more granular view of the result, as shown in Figure 3.8, where the indexes are the malware families (Table 3.1). On the other hand, the combination setups indicate accurate results compared with single content fingerprints. We depict three sub-fingerprints, which correspond to the best results. Note that the setup composed of *assembly*, *permission*, and *dex* byte-code shows the highest *F1-Score*.



(a) Assembly (F1-score=69%) (b) Permission (F1-score=69%) (c) Dex (F1-score=41%)

(d) Assembly, Permission, Dex, APK (F1-score=81%) (e) Assembly, Permission (F1-score=82%) (f) Assembly, Permission, Dex (F1-score=85%)

Figure 3.8: Confusion Matrices of Family-Fingerprint

Fingerprint Setup	*F1-Score*	*Precision*	*Recall*
Assembly	76%	88%	68%
APK	33%	36%	32%
Permission	76%	84%	70%
Dex	44%	46%	43%
Assembly, Permission, Dex, APK	83%	88%	80%
Assembly, Permission	84%	88%	81%
Assembly, Permission, Dex	**86%**	**89%**	**84%**
Best Fingerprint Setup	**86%**	**89%**	**84%**

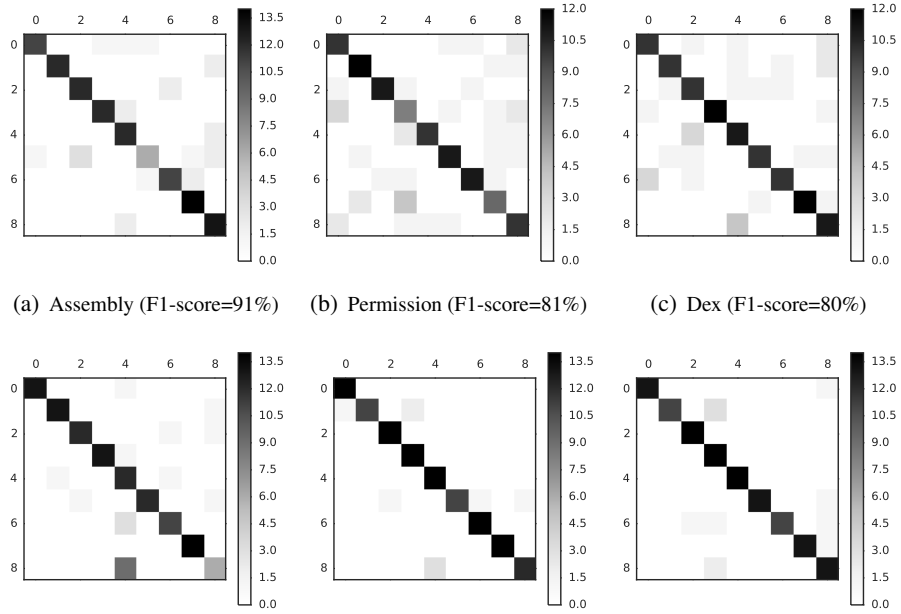
Table 3.2: Accuracy Results of the Family-Fingerprinting Approach

Peer-Matching Results. *Peer-matching* shows a higher F1-score, precision, and recall for all the setups compared to *family-fingerprinting*, as shown in Table 3.3. This can be clearly seen in the confusion matrices in Figure 3.9. In contrast to the previous results, the *dex* byte-code shows a higher *precision* than *assembly* and *permission*, but it is still lower in both *recall* and *F1-score*. The setup combination (*assembly*, *permission*) has the highest accuracy in the *peer-matching* approach. As such, using only two content categories, metadata *permission* and *assembly* instruction sequences, we achieve a very promising detection rate, especially considering that the computation of these

sub-fingerprints is light and simple compared to the state-of-the-art fingerprint hashing techniques.

Fingerprint Setup	*F1-Score*	*Precision*	*Recall*
Assembly	91%	91%	90%
Apk	46%	48%	44%
Permission	81%	82%	80%
Dex	86%	90%	84%
Assembly, Permission, Dex, APK	85%	91%	81%
Assembly, Permission, Dex	93%	94%	93%
Assembly, Permission	**94%**	**95%**	**94%**
Best Fingerprint Setup	**94%**	**95%**	**94%**

Table 3.3: Accuracy Result of Peer-Matching Approach



(a) Assembly (F1-score=91%) (b) Permission (F1-score=81%) (c) Dex (F1-score=80%)

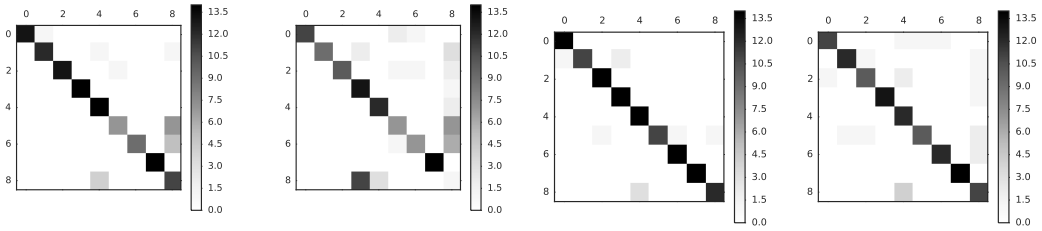
(d) Assembly, Permission, Dex, APK (F1-score=84%) (e) Assembly, Permission (F1-score=94%) (f) Assembly, Permission, Dex (F1-score=93%)

Figure 3.9: Confusion Matrices of Peer-Matching Approach

Peer-Voting vs Merged Fingerprints. As presented in Section 3.2.2, the most obvious technique to deal with multiple sub-fingerprints is to merge all of them (*merged fingerprint*). However, we propose *peer-voting* to compare multiple sub-fingerprints and use the majority voting to confirm the similarity. To test the proposed technique, we evaluate it against the *merged fingerprint* for the same fingerprinting setup. As shown in Table 3.4, *peer-voting* shows a higher accuracy than the merging one. A more illustrative view of the result can be seen in the confusion matrix in Figure 3.10.

Fingerprint Setup	*F1-Score*	*Precision*	*Recall*
Merged in Family-Approach	77%	84%	72%
Peer-Voting in Family-Approach	**85%**	**89%**	**84%**
Merged in Peer-Approach	87%	87%	86%
Peer-Voting in Peer-Approach	**94%**	**95%**	**94%**

Table 3.4: Accuracy Result Using Merged Fingerprint



(a) Voting in Family (85%) (b) Merged in Family (72%) (c) Voting in Peer-Matching (94%) (d) Merged in Peer-Matching (86%)

Figure 3.10: Confusion Matrices and F1-score using Merged Fingerprint and Peer-Voting

3.3 Malicious Community Fingerprints

3.3.1 Background

Usage Scenarios

In the context of this thesis research, Cypider has two primary usage scenarios. In the first scenario, Cypider can be applied only to malicious Android apps. The aim is to speed up the analysis process and attribute malware to their corresponding families. Under the first scenario, the overall malware analysis process is boosted by automatically identifying malware families and minimizing the overall manual analysis effort. The outcome of the previous process consists in the communities of malicious apps. The attribution of a family to a given community can be achieved by attributing a small set (one app in most cases) among its malicious apps. In the second scenario, Cypider is applied to mixed Android apps (i.e., malicious or benign). Such a dataset could be the result of a preliminary suspiciousness app filtering. Therefore, a lot of *false positives* can be recorded; we assume that benign apps - meaning *false positives* - constitute $50\% - 75\%$ of the actual suspicious apps. Based on the previous assumption, we could identify malicious Android apps by detecting and extracting app communities that share a common payload. We could infer

that apps with high similarity are most likely to be malicious.

3.3.2 Clustering Process

Cypider framework uses a dataset of unlabeled apps (malicious or mixed) in order to produce *community fingerprints* for the identified *app communities*. Cypider overall process is achieved by performing the following steps, as illustrated in Figure 6.1:

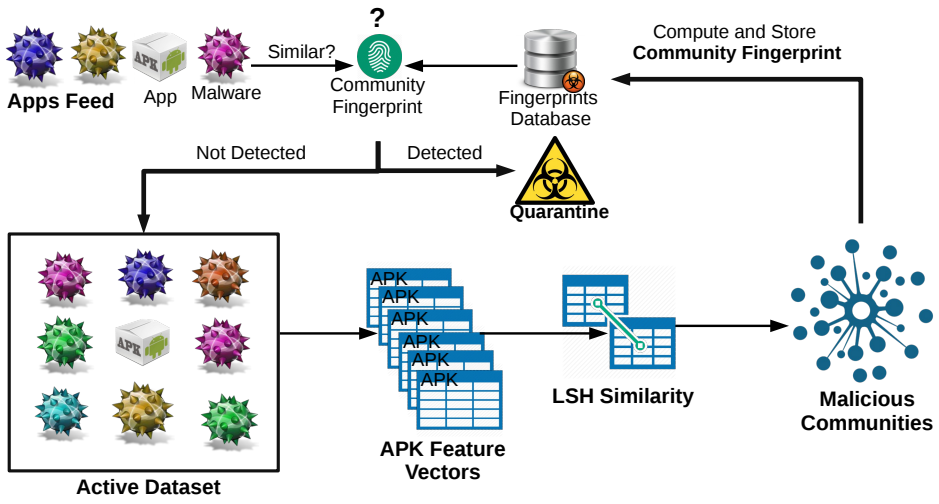


Figure 3.11: Cypider Framework Overview

At the beginning of the Cypider process, we need to filter out apps developed by the same author; we call them *sibling apps*. We aim here to remove the noise of having app communities of sibling apps because they tend to have many similar features since authors reuse components across different apps. Cypider identifies sibling apps in the dataset based on their version, app hash, and author cryptographic signature (provided in the META-INF directory in the APK file). Therefore, we only keep apps with no duplication in the author identities since adversaries favor the use of multiple fake author identities to prevent the removal of all apps in case of detected maliciousness in one of them. Regarding numerous apps with the same author, Cypider randomly selects one app. Afterward, if the chosen app is recognized as malicious in the analysis results, Cypider will tag all its sibling apps as malicious.

After filtering the sibling apps, we need to derive from the actual app's package meaningful information that could identify the app and help to compute the similarity against other apps. For this

purpose, Cypider extracts static features from the apps, which could be either benign or malicious, depending on the usage scenario. Feature engineering is the most critical part of the whole framework in the context of Android malware detection (other usage scenarios could have different static features, but the overall approach is the same). It is essential to mention that the selected features must be resilient to the attacker's deceiving techniques. To this end, the features need to be broad enough to cover most of the static characteristics of a given Android APK. The more extensive the features are, the more resilient they are. For our purposes, we leverage static analysis features of the APK in the design of Cypider. In particular, we extract such features from each content category (classes.dex, resources, assembly, etc.), as described in Section 3.3.3.

Relying on the extracted features from each content, Cypider computes a fixed-length *feature vector* for each content features. In order to reduce and normalize the size of the feature vectors, we equip Cypider with a machine learning preprocessing technique, namely *feature hashing* [173] (or *hashing trick*), as presented in Section 3.3.3. As a result, Cypider produces, from the extracted features of the previous stage, multiple feature vectors with a small and fixed size. The number of the generated feature vectors depends on how many APK contents are used in feature extraction (each content type corresponds to one feature vector).

For efficient comparison between apps, we empower Cypider system with a highly scalable similarity computation system based on locality-sensitive hashing (LSH) [80], which computes the similarities between apps, as presented in Section 3.3.4. Given a pair of apps, we calculate the similarity between each content *feature vector* from the previous stage to decide if they are connected or not from the perspective of to that content. The result of this step is an undirected network (or similarity network), where the nodes are Android apps, and the edges represent the high similarity to one content between apps. For similar apps, multiple connecting edges are expected. Besides, the more edges are, the more the apps are suspected to be malicious.

Cypider leverages a *similarity network* in order to detect malicious app communities. For malicious app, Cypider extracts highly connected app communities and then excludes these apps from the dataset. The remaining apps (i.e., apps that are not part of any community) are considered in another Cypider malware detection iteration. We expect to get a pure (only malware apps from the same family) or near-pure community if the containing apps of a given community have respectively

the same or almost the same Android malware family. In the case of a mixed dataset, Cypider first excludes all the app nodes with a degree 1 (i.e., the app is only self-similar), which are most likely to be benign apps (or might be a zero day threat). Afterward, Cypider extracts the apps of malicious communities.

The rest of apps will be considered in another Cypider iteration. At this point, we expect to have some benign communities as *false positives*. However, the similarity network makes Cypider's decision explainable (the security practitioner can check which static contents are similar between the detection communities) because the security practitioner can track which content these apps are similar to. The previous option could also help sharpening the static features to prevent benign apps from being detected in malicious communities. For community detection (Section 5.2.1), we adopt a highly scalable algorithm [82] to enhance Cypider's community detection module.

To this end, we consider a set of malicious communities, each of which is most likely to be a malware family or a subfamily. Cypider leverages these malicious communities to generate the so-called *community fingerprint* (Section 3.3.6) that captures the app features of a given detected community. Instead of using traditional crypto or fuzzy hashing of only one malware instance, we leverage a model produced by a one-class classifier [170], which provides a better-compressed format of a given Android malware family. This model is used to decide whether new malware apps are part of this family or not. The results consist of multiple *community fingerprints*, each of which corresponds to a detected community. The generated fingerprints are stored in the *signature database* for later use.

To this end, Cypider is ready to start another detection iteration with a new dataset, including the rest of unassigned apps from the previous iteration. The same previous steps will be followed for the new iteration. However, at this point, Cypider first checks the *feature vectors* of the new apps against the known *malware community fingerprints* stored in the database. The matched apps to a community fingerprint are labeled as malicious without adding them to the *active dataset*. Undetected apps are added to the *active dataset* and are considered in the next iteration of the detection process.

We consider Cypider approach as a continuous process, in which we detect and extract communities from the *active dataset* that always gets new apps (malware only or mixed with benign)

daily, in addition to the rest of apps from the previous iterations.

3.3.3 Static Features

In this section, we present static features of Android packaging (*APK*). In this chapter, we only extract features from each app *APK* file using static analysis, to generate feature vectors, which are then used to compute the similarity with other apps feature vectors. In other words, the feature vector set will be the input to the LSH similarity computation module used to build the similarity network. As previously mentioned, the features should be broad enough to cover most of the static content of the APK file. Features could be categorized according to the main APK content types to: i) Binary features, which are related to bytecode (Dex file) of the Dalvik virtual machine considering the hex dump of the Dex file along with the actual file. ii) Assembly features, which are computed from the assembly of *classes.dex*. iii) Manifest features, extracted from the Manifest file, which is vital to Android apps since it provides essential information about the app to the Android OS. iv) *APK features*, which include all the remaining APK file content, such as *resources* and *assets*. In this section, we present the static features based on the adopted concept to extract them (e.g., N-gram).

N-grams.

The *N-gram* technique is used to compute contiguous sequences of N items from a large sequence. For our purpose, we use N-grams to extract the sequences derived from Android malware content with the aim to discriminate different malware samples. The N-grams from various Android app package contents, such as *classes.dex*, reflect the APK patterns and implicitly capture the underlying Android package semantics. We compute multiple feature vectors for each APK content. Each vector $V \in D$ ($|D| = \Phi^N$ where Φ represents all the possibilities of a given APK content). Each element in the vector V contains the number of occurrences of a particular APK content N-gram.

Classes.dex Byte N-grams. To increase the extracted information, we leverage two types of N-grams, namely *opcode N-grams* and *byte N-grams*, which are extracted from the binary *classes.dex*

file and its assembly respectively. From the *hexdump* of the *classes.dex* file, we compute *Byte N-grams* by sliding a window of the hex string as depicted in Figure 3.4.

Assembly opcodes N-grams. The opcode N-grams are unique sequences in the disassembly of *classes.dex* file, where the instructions are stripped from their operands. We choose opcodes instead of full instructions for multiple reasons: (1) Using opcodes tends to be more resilient to simple obfuscations that modify some operands such as hard-coded IPs or URLs. (2) Opcodes could be more robust to modifications, caused by repackaging, that alter or rename some operands. (3) In addition to being resilient to changes, opcodes can be efficiently extracted from Android apps.

The gained information from opcode N-grams could be increased by considering only functions that use a sensitive APIs such as SMS API. Also, excluding the most common opcode sequence decreases the noise in N-gram information. Also, the number of N-grams has a significant influence on the gathered semantics. The result of N-gram extraction is the list of unique N-grams with the occurrence number for each content category, i.e., *opcode instructions*, *classes.dex*. In addition to the opcodes, we also consider the *class names* and the *methods' names* as assembly features.

Native Library N-grams

The *Native Library* is part of the binary sub-fingerprint, which captures C/C++ shared libraries [19] used by malware. Using the native library for malware fingerprinting is essential in some cases to distinguish between two Android malware samples. For instance, if the malware uses a native library, it is more likely to be **DroidKungFu2** than **DroidKungFu1** because **DroidKungFu2** malware family uses C/C++ library and **DroidKungFu1** uses only *Java bytecode*.

APK N-grams. The N-gram of the APK file can give an overview of the APK file semantics. For instance, most of the repackaged apps are built from an original app with minor modifications [99]. Consequently, applying N-gram analysis on the APK file can detect a high similarity between the repackaged app and the original one. Besides, some components of the APK file, e.g., images and GUI layout structures, are preserved by the adversaries, especially if the purpose of the repackaging process is to develop a phishing malware. Both apps, in this case, are visually similar, and hence

the N-gram sequences computed from both apps will be similar in the zone related to the resource directory.

Manifest File Features

In our context, *AndroidManifest.xml* is a source of essential information that could help in identifying malicious apps. The *permissions* required by apps are the most important features. For example, apps that require *SMS send* permission are more suspicious than other apps since a big portion of Android malware apps target sending SMS to premium charging phone numbers. In addition, we extract other features from *AndroidManifest.xml*, namely, *activities*, *services*, and *receivers*.

Android API Calls

The required permissions provide a global view of possible app behaviors. However, we could get a more granular view by tracking *Android API calls*, knowing that one permission could allow access to multiple *API calls*. Therefore, we consider the *API* list used by the apps as the feature list. Furthermore, we use a filter list of *API* of the *suspicious APIs*, such as *sendTextMessage()* and *orphan APIs*, which are part of an undeclared permission. On the other hand, we extract the list of permissions, where none of their *APIs* has been used in the app.

Resources

In this category, we extract features related to *APK* resources, such as text strings, file names, and their content. An important criterion when filtering the files is to exclude the names of standard files, e.g., *String.xml*. Also, we include files' contents by computing **MD4** hashes on each resource file. At first glance, it seems that the use of MD4 is not convenient compared to more modern cryptographic hashing algorithms such as MD5 and SHA1. However, we choose the MD4 purposely because it is cheap in terms of computation. This allow to enhance the scalability of the system, yet, we achieve the goal of the file comparison between the malicious apps of the active dataset. Finally, we make a text string selection in the text resources, where we leverage *tf-idf* (term frequency-inverse document frequency) [190] technique for this purpose.

APK Content Types

Table 3.5 summarizes the proposed feature categories based on APK contents. It also depicts the features considered in the current implementation of Cypider We believe that the used features give a more accurate representation of Android packages as we showed in Section 3.2 . On the other hands, the features that we excluded such as *Text Strings* and *Assembly Class Names* are highly vulnerable to common obfuscation techniques. Also, excluded features such as *Manifest Receivers* generate very sparse features vectors, which effect the overall accuracy.

#	Content Type Features	Implemented Feature
0	APK Byte N-grams	X
1	Classes.dex Byte N-grams	X
2	Native Library Bytes N-grams	X
3	Assembly Opcodes Ngrams	X
4	Assembly Class Names	
5	Assembly Method Names	
6	Android API	X
7	Orphan Android API	
8	Manifest Permissions	X
9	Manifest Activities	X
10	Manifest Services	X
11	Manifest Receivers	
12	IPs and URLs	X
13	APK Files names	X
14	APK File light hashes (md4)	
15	Text Strings	

Table 3.5: Content Feature Categories

Feature Preprocessing

Feature extraction and similarity computation are the core operations in the proposed framework. Therefore, we need to optimize both their design and implementation to get the intended scalability. The expected output from *feature processing* is a vector, which can straightforwardly be used to compute the similarity between apps. App feature vectors are the input to Cypider community detection system.

The N-gram technique, presented in Section 3.3.3, suffers from its very high dimensionality D. The dimension number D hyper-parameter dramatically influences the computation and the memory needed by Cypider for Android malware detection. The complexity of computing the extracted N-grams features increases exponentially with N. For example, for the *opcodes N-grams*, described in Section 3.3.3, the dimension D equals to R^2 for bi-grams, where $R = 200$, the number

of possible opcodes in Dalvik VM. Similarly, for *3-grams*, the dimension $D = R^3$; for *4-grams*, $D = R^4$. Furthermore, N has to be at least 3 or 5 to capture the semantics of some Android APK content.

To address this issue, we leverage the *hashing trick* technique [173] to reduce the high dimensionality of an arbitrary vector to a fixed-size feature vector. More formally, the *hashing trick* reduces a vector V with $D = R^N$ to a compressed version with $D = R^M$, where $M << N$. The compacted vector boosts Cypider, both computation-wise and memory-wise, by allowing the clustering system to handle a large volume of Android apps. Previous research [173, 188] has shown that the hash technique could preserve a decent amount of information in the vector distance. Moreover, the computational cost incurred by using the hashing technique for reducing dimensionality grows linearly with the number of samples and groups. Algorithm 1 illustrates the overall process of computing the compacted feature vector from an N-grams set. Furthermore, it helps to control the length of the compressed vector in an associated feature space.

3.3.4 LSH Similarity Computation

Building the *similarity network* is the backbone of Cypider framework. We generated the *similarity network* by computing the pair-wise similarity between each feature vector of the apps APKs. As a result, we obtain multiple similarities according to the number of these content vectors. Using various similarities gives flexibility and modularity to Cypider. In other words, we could add any new feature vector to the similarity network without disturbing Cypider process. Also, we could remove features without affecting the overall process, which makes the experimentation of selecting the best features more convenient. More importantly, having multiple similarities between apps static contents in the similarity network leads to explainable decisions, where the investigator can track which contents a pair of apps are similar in the final similarity network. Similarity computation needs to be conducted in an efficient way that is much faster than the brute-force computation. For this purpose, we leverage LSH techniques, and more precisely *LSH Forest* [80], a tunable high-performance algorithm for similarity computation, employed by the Cypider framework. The key idea behind *LSH Forest* is that similar items hashed using *LSH* are most likely to be in the same

bucket (collide) and dissimilar items will be in different ones. Many similarity measures correspond to *LSH* function with this property. In our case, we use the well-known *Euclidean* distance for this purpose.

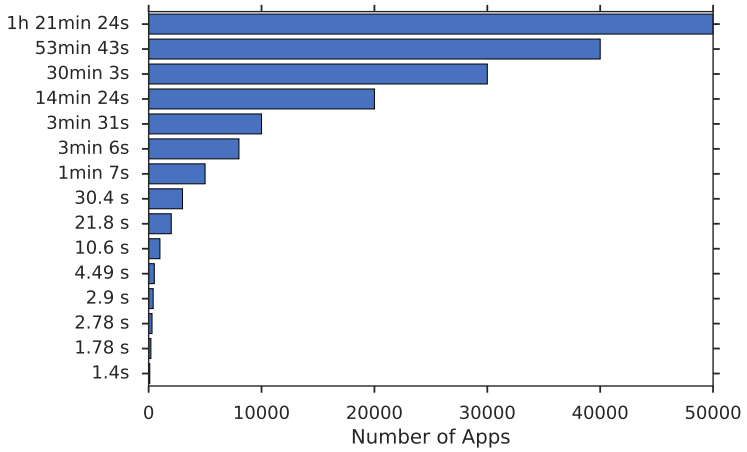


Figure 3.12: LSH Similarity Computational Time

$$d(m,n) = \|V_m - V_n\| = \sqrt{\sum_{i=1}^{|S|}(V_m(i) - V_n(i))^2} \tag{3}$$

Given a pair of Android apps, after extracting one content feature vector, we use the *Euclidean* distance to compute the distance between two feature vectors m and n of one APK content, as depicted in Equation 3. Figure 3.12 shows the LSH computation time with respect to the number of apps using *one CPU core* and *one thread* for the permission feature vector. Even though the current performance using *LSH Forest* is acceptable for a large number of daily malware samples (reaching $40,000$ apps per hour), we believe that we could drastically improve these results by just leveraging an implementation that exploits all CPU cores in addition to multi-threading. The final result of the similarity computation is a *heterogeneous network*, where the nodes are the apps, and edges represent similarities between apps if a certain threshold is exceeded. The **similarity threshold** is the percentage of **average similarity** of a given content. In other words, we compute the average value of all the pairwise similarities for each feature content. Afterward, we set a percentage from this average to be the final threshold. We use **similarity threshold** for all feature contents even though they have different average values. The **similarity threshold** is systematically fixed based

on our evaluation, and the same threshold is used in all experiments. However, we investigate the effect of the **similarity threshold** on Cypider performance in the evaluation section. Note that the network is heterogeneous because there are multiple types of edges, where the edge type represents static content type.

3.3.5 Community Detection

A scalable community detection algorithm is essential to extract *suspicious communities*. For this reason, we empower Cypider with *Fast unfolding community detection* algorithm [82], which can scale to billions of network links. The algorithm achieves excellent results by measuring the *modularity* of communities. The *modularity* is a scalar value $M \in [-1, 1]$ that measures the density of edges inside a given community compared to the edges between communities. The algorithm uses an approximation of *modularity* since finding the exact value is computationally hard [82]. The previous algorithm requires a homogeneous network as input to work properly.

For this reason, we propose using a *majority-voting* mechanism (Section 3.2.1) to homogenize the heterogeneous network generated by similarity computation. Given the number of content similarity links s, the *majority-voting* method decides whether a pair of apps are similar or not by computing the ratio s/S, where S is the number of all contents used in the current Cypider configuration. If the ratio is above the average, the apps will only have one link in the *similarity network*. Otherwise, all the links will be removed. Notice that content similarity links could be retained for later use, for example, to conduct a thorough investigation about given apps to figure out how similar they are, and on which content they are similar. The prior use case could be of a great importance for security analysts.

We propose a *majority-voting* mechanism (see Section 3.2) to filter links between the nodes (apps) to prevent having inaccurate *suspicious communities*. Furthermore, we employ a *degree filtering* hyper-parameter to filter all node links with a degree that is less than a threshold value. The previous hyperparameter keeps only edges of a given node when their number is above the threshold. We call this hyper-parameters a **content threshold**, which is the number of similar contents to keep a link in the final similarity network.

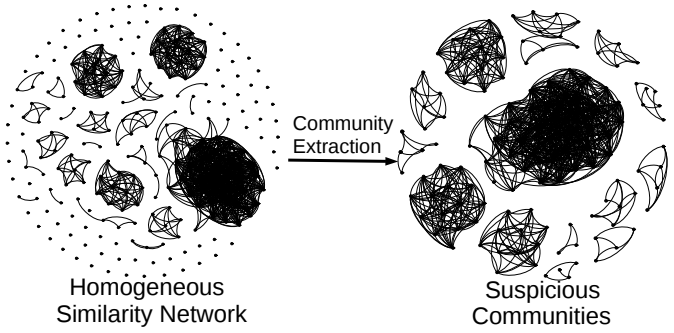


Figure 3.13: Applying Cypider on a Small Dataset

Consequently, only nodes with high connectivity will maintain their edges, which are supposedly similar to malicious apps. Notice that all parameters have been fixed in our evaluations. In the case of a *mixed dataset* scenario, we use the degree 1 to filter all apps having a similarity link to themselves since they are not similar to any other app in the *active dataset*. Cypider filters these apps and consider them as benign apps. At this point, Cypider applies the community detection algorithm [82] to extract a set of communities with different sizes. Afterward,all communities are filtered with a community cardinal that is less than the *minimum community size* parameter (fixed for all the evaluations). The purpose of the filtrating is to prevent the extraction of bad quality communities. Figure 3.13 depicts an example of using Cypider on a small Android dataset (250 malware apps), where the process of community detection starts with *homogeneous network* and ends up with *suspicious communities*. The **content threshold** and **community size** hyper-parameters are thoroughly investigated in the evaluation section.

3.3.6 Community Fingerprint

Finding malicious communities is not the only goal of Cypider framework. Since Cypider is completely unsupervised, we aim at generating fingerprints from the extracted communities automatically. Therefore, in the following iteration, Cypider filters known apps without adding them to the *active dataset*. Traditional *cryptography fingerprints* or *fuzzy hashing* techniques are not suitable for our case since we aim to generate a fuzzy fingerprint, not only for one app but also for the whole malicious community whether it is a malware family or subfamily. In this context, we present a novel fingerprinting technique using the *One-Class Support Vector Machine* learning

model (OC-SVM) [170], which was found to be fuzzy enough to cover malicious community apps. The *One-Class SVM* model is used to detect the set of a set of malicious apps of a given community. In particular, it detects the *soft boundary model* of that set. Adopting the *one-class model*, Cypider classifies new apps as belonging to this community or not. The proposed fingerprinting technique produces a much more compressed signature database compared to the traditional methods, where the signature is generated only for one malicious app. Moreover, it notably reduces the computation since we check only the community fingerprints instead of checking each single malware hash signature. In order to generate a *community fingerprint* from a set of malicious apps, Cypider first extracts the features, as presented in Section 3.3.3. Afterward, the *one-class model* is trained using static features of malicious community apps.

3.3.7 Experimental Results

In this section, we present Cypider implementation and the testing setup, including the dataset and the performance measurement techniques. Afterward, we present the achieved results regarding the defined metrics for both usage scenarios that are adopted in Cypider framework, namely *malware only* (only malicious apps) and *mixed* (benign and malicious apps) datasets.

Dataset and Test Setup

In order to evaluate Cypider, we leverage well-known Android datasets, namely, (i) MalGenome malware dataset [17, 203], (ii) Drebin malware dataset [18, 73, 175], and AndroZoo public Android app repository. As presented in Table 4.3, we produce two other evaluation datasets based on the previous ones by adding Android apps downloaded from Google Play in late 2014 and the beginning of 2019. These apps have been randomly downloaded without considering their popularity or any other factor. In order to build *Drebin Mixed* and *AndroZoo Mixed* datasets, we added $4,403$ benign apps to the original *Drebin* dataset. The result is a mixed dataset (malware and benign) with 50% of apps in each category. Similarly, we build the *MalGenome Mixed* dataset with 75% of benign apps.

The aim of using these datasets is to evaluate Cypider in unsupervised usage scenarios, with and without benign apps. First, we assess Cypider on malware only using *Drebin*, *AndroZoo*, and *Genome* datasets. This use case is the most attractive one in bulk malware analysis since it decreases

	Drebin	*DrebinMixed*	*Genome*	*GenomeMixed*	*AndroZoo*	*AndroZooMixed*
Size	4330	8733	1168	4239	66k	66k
Malware	4330	4330	1168	1168	66k	66k
Benign	0	4403	0	3071	0	44k
Families	46	46	14	14	-	110k

Table 3.6: Evaluation Datasets

the number of malware to be analyzed by considering only a sample from each detected community. Second, *Cypider* is evaluated against mixed datasets. The second scenario is more challenging because we expect not only *suspicious communities* as output but also benign communities (false positives) along with filtered benign apps.

To asses Cypider obfuscation resiliency, we conduct the evaluation on PRAGaurd obfuscation dataset[2], which contains $11k$ obfuscated maclicious apps using common obfuscation techniques [147]. In addition, we generate $100k$ benign and malware obfuscated apps using DroidChameleon obfuscater [162] using common obfuscations techniques and related combinations.

To this end, various metrics are needed to measure Cypider performance in each dataset. We adopted the flowing metrics:

App Detection Metrics

A1: *True Malware*: This metric computes the number of malware apps that are detected by Cypider. It is applied to both usage scenarios.

A2: *False Malware*: This metric computes the number of benign apps that have been detected as a malware app. It is applied only to the *mixed dataset* since there are no benign apps in the other datasets.

A3: *True Benign*: This metric computes the number of filtered benign apps by Cypider. It is only applied to *mixed dataset* evaluation.

A4: *False Benign*: This metric computes the number of malware apps that are considered as benign in the *mixed dataset* evaluation.

[2]http://pralab.diee.unica.it/en/AndroidPRAGuardDataset

A5: *Detection Coverage*: It measures the percentage of the detected malware from the overall dataset. Formally, it is the number of clustered Android apps divided by the total number of apps in the input dataset.

Community Detection Metrics

C1: *Detected Communities*: It indicates the number of suspicious communities that have been extracted by Cypider.

C2: *Pure Detected Communities*: This metric computes the number of communities with a unique Android malware family. In other words, a community is pure if it contains instances of the same family. In this task, we rely on the labels of the used datasets to check the purity of a given community. This metric is applied to both usage scenarios.

C3: *K-Mixed Communities*: This metric counts the communities with K-mixed malware families, where K is the number of families in a detected community. This metric is applied to both usage scenarios.

C4: *Benign Communities*: This metric computes the number of benign communities that have been detected as suspicious. This metric is applied to in the *mixed dataset* evaluation.

Mixed Dataset Results

Table 3.7 presents the evaluation results of Cypider using *Drebin Mixed* and *Genome Mixed* datasets. The most noticeable result is the fact that Cypider detects about *half* of the actual malware in a single iteration in both datasets even though the noise of benign apps (false positive) is about 50% to 75% of the actual dataset. On the other hand, Cypider is able to filter a considerable number of benign apps from the dataset. However, in both dataset evaluations, we obtain some *false malware* ($190 - 103$ apps) and *false benign* ($38 - 10$ apps) respectively to datasets. According to our results, these *false positives,* and *false negatives* appear, in most cases, in communities with the same labels (malware or benign). Therefore, the investigation would be straightforward by analyzing some samples from a given suspicious community. The similarity network and the resultant communities are illustrated in Figure 3.15(a) and Figure 3.15(b) respectively.

Community Metrics	**Drebin Mixed**	**Genome Mixed**
True Malware A1	2413	449
False Malware A2	190	103
True Benign A3	257	171
False Benign A4	38	10

Table 3.7: Mixed Evaluation Using Apps Metrics

Table 3.8 presents the results of Cypider's evaluation using *community metrics*. A very interesting result here is the number of *pure detected communities*, which is 179 pure communities out of 188 detected communities in *Mixed Drebin* and 61 pure communities out of 61 detected communities (perfect purity) in *Mixed Genome*. Consequently, almost all the detected communities have instances in the same malware family or benign ones. Even the *mixed communities* are composed of only two labels (2-mixed). It is important to mention that all the *detected benign communities* are pure without any malware instance, which makes the investigation much easier. Furthermore, according to our analysis, most malware labels in the *2-mixed* malicious communities are just a naming variation of the same malware, which is caused by name convention differences among vendors. For example, in one *2-mixed* community, we found *FakeInstaller* and *Opfake* malware instances. Actually, these names point to the same malware [13], which is *FakeInstaller*. Similarly, we found *FakeInstaller* and *TrojanSMS.Boxer.AQ*, which points to the same malware [46] with different vendor naming.

Apps Metrics	**Drebin Mixed**	**Genome Mixed**
Detected C1	188	61
Pure Detected C2	179	61
2-Mixed C3	9	0
Benign C4	18	16

Table 3.8: Evaluation Using Community Metrics

Community Metrics	**Drebin**	**Genome**
True Malware A1	2223	449

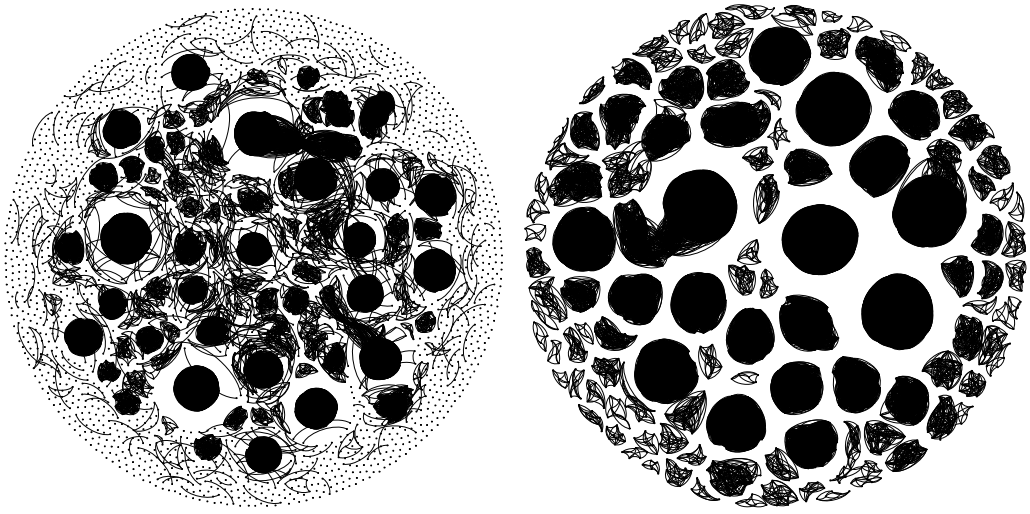
Table 3.9: Malware Evaluation Using Apps Metrics

Apps Metrics	**Drebin**	**Genome**
Detected C1	170	45
Pure Detected C2	161	45
2-Mixed C3	9	0

Table 3.10: Evaluation Using Community Metrics

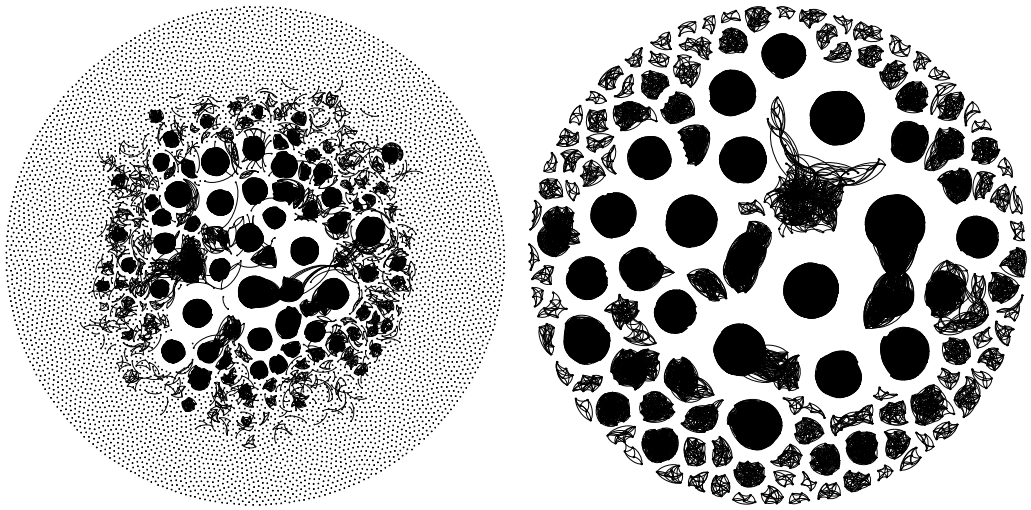
Results of Malware-only Datasets

Tables 3.9 and 3.10 present the performance results of Cypider using the *app metrics* and *community metrics* on malware only datasets. Since we use the same malware dataset as the *mixed dataset* by only excluding benign apps, we obtain almost the same results. Cypider was able to detect about 50% of all malware in one iteration. Moreover, nearly all the recognized communities are pure. This high quality result is a significant advantage of Cypider in malware investigation since the security analyst could automatically attribute the family to a given suspicious community could be by only matching one or two samples. Furthermore, the analysis complexity dramatically decreases from $2,413$ detected malware to only 188 discovered communities. We believe that this could reduce the analysis window and help overcome the overwhelming number of daily detected Android malware. Notice that there are nine *2-mixed* communities in the *Drebin dataset*, which contain different malware labeled for the same actual malware, as mentioned before. Figure 3.14(a) depicts the *similarity network* of the Drebin malware dataset. After applying the community detection algorithm, we end up with *malicious communities*, as depicted in Figure 3.14(b).



(a) Simialrity Network (b) Detected Communities

Figure 3.14: Cypider Network of Drebin Malware Dataset



(a) Simialrity Network (b) Detected Communities

Figure 3.15: Cypider Network of Drebin Malware Dataset

Family	Community Size	Family Detection (Acc)	Malware (Acc)	Benign (Acc)	General (F1)
HiddenAds	1774	77.43	93.40	94.93	87.83
BridgeMaster	1041	76.99	93.08	93.22	87.05
InfoStealer	2973	86.60	88.37	91.26	75.50
Plankton	495	77.76	100.0	100.0	75.10
BaseBrigge	1499	76.76	88.51	92.47	73.28
Utchi	973	76.19	99.98	100.0	72.02

Table 3.11: Community Fingerprint Accuracy on Different Families

Community Fingerprint Results

Table 3.11 shows the evaluation results with respect to *community fingerprinting*, which is applied to different detected communities with various Android malware families. The community fingerprint model (One-Class SVM) achieves 87% F1-score in detecting malware from the same malware family that is used in the training phase. In the *signature database*, these new malware samples share the same family with a given *community fingerprint*. Furthermore, the compressed format of this fingerprint, i.e., learning model in a binary format, could fingerprint an entire Android family, which generated a significantly more compacted *signature database*.

The performance of the community fingerprint mainly depends on the number of malware in the detected community. Higher detection performance is achieved when more malware instances exist in the community. In this respect, we determine a threshold systematically for the community

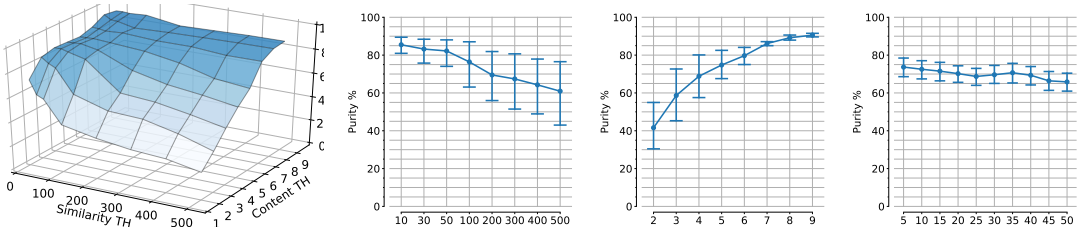
cardinality (size), which is required to compute the fingerprint and store it in the *signature database*. As shown in Table 3.11, one of the main characteristics of community fingerprinting, is its ability to differentiate between general malware and benign samples with high accuracy. The reason behind this high accuracy is the high similarity between general malware and the trained family. Notice that the one-class SVM model is trained on samples from only one malware family. In other words, malicious apps tend to have similar features, although they do not belong to the same malware family. Thus, benign samples and general malware are highly dissimilar, and hence benign samples are less likely to match with community fingerprint, which minimizes the overall false positive.

3.4 Hyper-Parameter Analyses

In this section, we analyze the effect of the employed hyper-parameters on Cypider on the overall performance measured using **Purity**, **Coverage**, and **Community Numbers** metrics. Specifically, we investigate the **similarity threshold**, the **content threshold**, and the **community size**, as presented in Section 3.3.4 and Section 5.2.1.

3.4.1 Purity Analysis

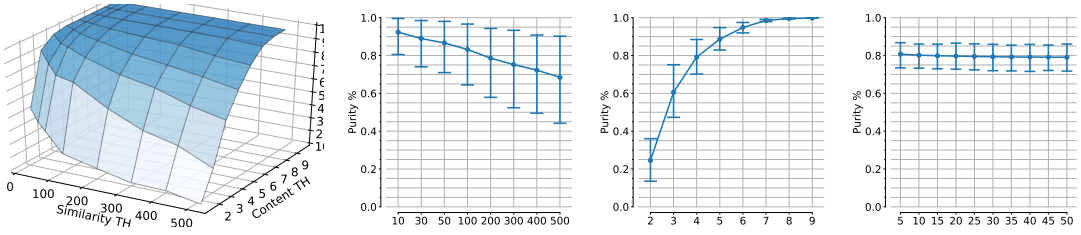
In the purity analysis, we compute the overall percentage of clustered malware samples of the groups belonging to the same Android malware family. A perfect purity metric means that each detected community (cluster) contains samples from the same Android malware family. Figures 3.16 and 3.17 show the effect of Cypider hyper-parameters on the purity of the detected malware communities in the similarity network of *Drebin* and AndroZoo datasets respectively.



(a) Similairty and Content (b) Similarity Threshold (c) Content Threshold (d) Community Size

Figure 3.16: Purity Hyper-Parameters on Drebin

It is worth noting that the **content threshold** is the most affecting hyper-parameter on the overall purity. A small content threshold results in a lower purity percentage, as shown in the evaluation of both *Drebin* and *AndroZoo* datasets. This finding is intuitive because Cypider grouping outcome is more accurate when using more content types threshold in the *majority voting* similarity computation.



(a) Similairty and Content (b) Similarity Threshold (c) Content Threshold (d) Community Size

Figure 3.17: Purity Hyper-Parameters on AndroZoo

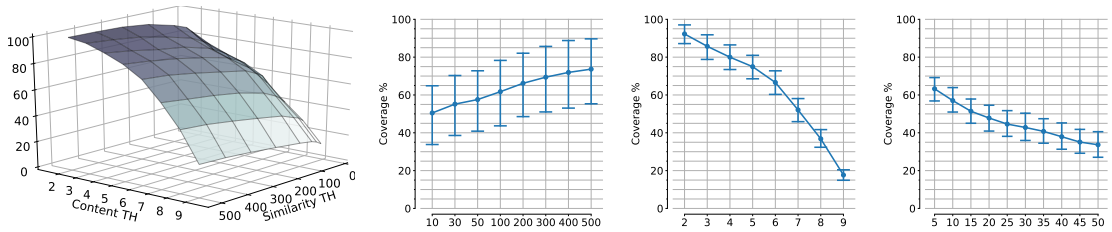
On the other hand, the **similarity threshold** has a secondary effect compared to the **content threshold**. This means that a tight threshold outputs less false samples in the detected communities. Finally, we notice a very minor effect of the **community size** on the overall purity metric for both *Drebin* and *AndroZoo* evaluations, as shown in Figures 3.16 and 3.17 respectively.

3.4.2 Coverage Analysis

In the coverage analysis, we assess the percentage of the detected malware from the overall input dataset. A perfect coverage means that Cypider detects malware in the produced malware communities. Figure 3.18 and 3.19 depict the change in the coverage percentage with Cypider hyper-parameters for *Drebin* and *AndroZoo* datasets respectively.

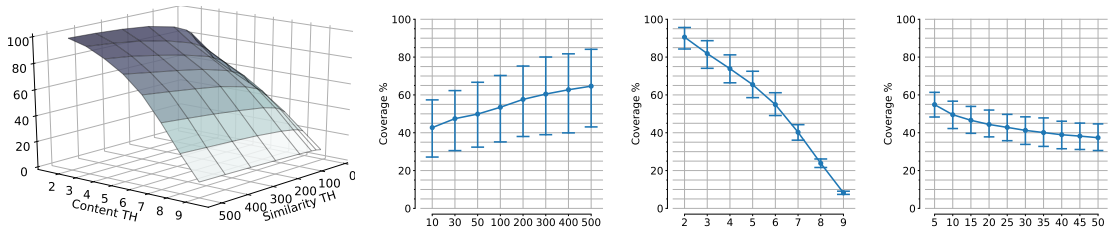
We notice that the **content threshold** is the most affecting hyper-parameter on the overall coverage metric. This means that a high content threshold in the majority voting (Section 3.3.4) leads to the detection of fewer malware samples in the produced malware communities. Therefore, the coverage metric decreases drastically with a high content threshold, as shown in Figures 3.18 and 3.19.

The **similarity threshold** and the **community size** have a secondary effect on the coverage



(a) Similairty and Content (b) Similarity Threshold (c) Content Threshold (d) Community Size

Figure 3.18: Coverage Hyper-Parameters on Drebin



(a) Similairty and Content (b) Similarity Threshold (c) Content Threshold (d) Community Size

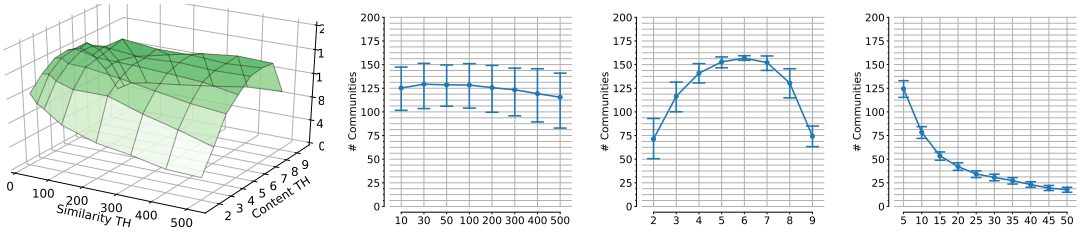
Figure 3.19: Coverage Hyper-Parameters on AndroZoo

metric. For the similarity threshold, a wide distance threshold yields a higher detection rate, and therefore a high coverage metric. For the community size threshold, a larger value leads to ignore many small malware communities, which affects the detection coverage metric negatively.

3.4.3 Number of Communities Analysis

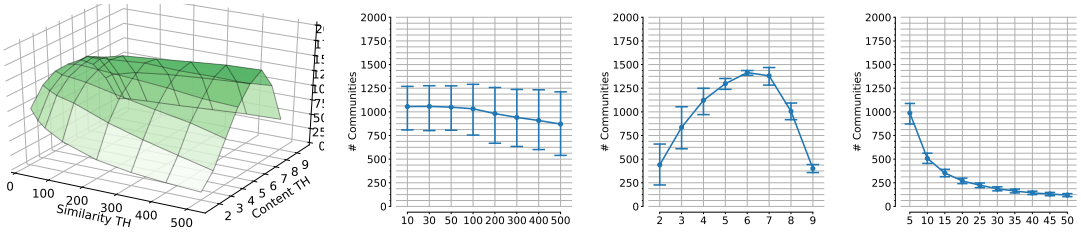
In this section, we analyze the total number of the detected communities produced by Cypider. A perfect Cypider clustering yields a result in which the number of communities is equal to the actual number of malware families in the input dataset. Figures 3.20 and 3.21 depict the effect of Cypider hyper-parameters on the number of the detected communities on *Drebin* and *AndroZoo* datasets respectively.

It is essential to mention that the community size has a strong influence on the number of communities. A higher community size threshold will filter many small malware communities, which influences the number of detected communities directly.



(a) Similairty and Content (b) Similarity Threshold (c) Content Threshold (d) Community Size

Figure 3.20: Detected Communities Hyper-parameters on Drebin



(a) Similairty and Content (b) Similarity Threshold (c) Content Threshold (d) Community Size

Figure 3.21: Detected Communities Hyper-parameters on AndroZoo

For the content threshold, the majority voting with a small content threshold causes many communities to merge. This is because the samples have to be similar in only two content thresholds to maintain a similarity. On the other hand, the majority voting with a high content threshold will detect fewer malware samples and, hence, less overall malicious communities. Finally, we notice a minor effect of the similarity threshold on the overall number of the detected communities, as depicted in Figures 3.20 and 3.21 for *Drebin* and *AndroZoo* datasets respectively.

3.4.4 Efficiency Analysis

In this section, we investigate the overall efficiency of the Cypider framework. Specifically, we present the runtime in seconds on the core computation of our framework: (i) the **similarity computation** to build the similarity network, (ii) the **community detection** to partition the similarity network into a set of malicious communities.

Figure 3.22 depicts the similarity computation time in seconds to build Cypider similarity network. We notice that Cypider framework is very efficient at producing the similarity network

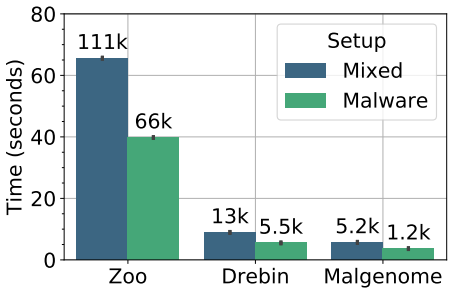
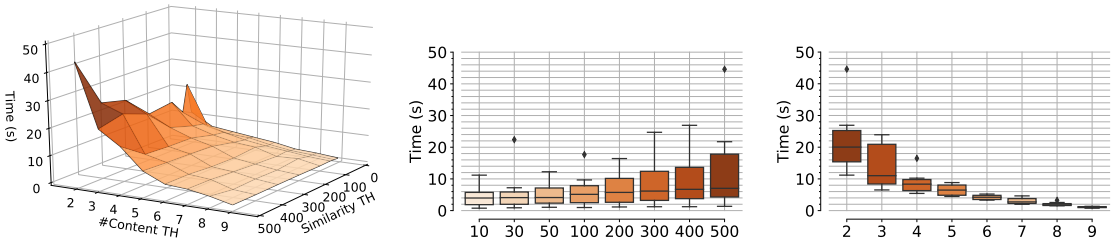


Figure 3.22: Similarity Computation Time

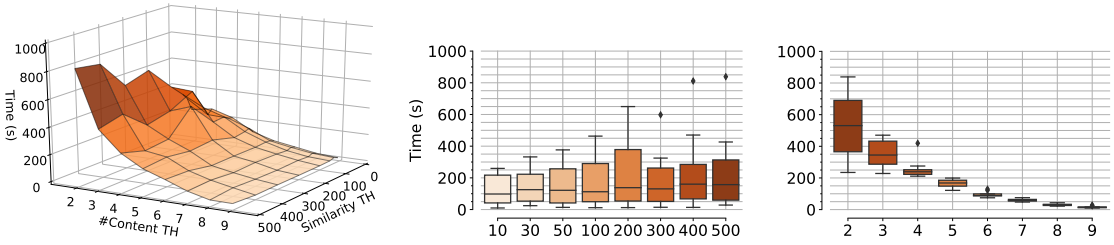
because we employ locality sensitive hashing techniques to speed up the pairwise similarity computation between the feature hashing vectors. For example, Cypider took only about 1 second to compute the similarity between $111k$ samples feature hashing vectors.



(a) Similairty and Content (b) Similarity Threshold (c) Content Threshold

Figure 3.23: Community Detection Time Analysis on Drebin

Figures 3.23 and 3.24 present the community detection time in seconds on *Drebin* and *AndroZoo* datasets respectively. We analyze the effect of *similarity* and *content* thresholds on the overall community detection time. In Figures 3.23 and 3.24, we notice that Cypider spends more time due to decreasing the content threshold and decreasing the similarity threshold in *Drebin* and *AndroZoo* experiments. The previous thresholds setup increases the density of Cypider similarity network, and therefore, the community detection processing takes more time in the partition process of the network. On the other hand, increasing the content threshold while decreasing the similarity threshold produces a very sparse similarity network, which takes a negligible time in the partition process.

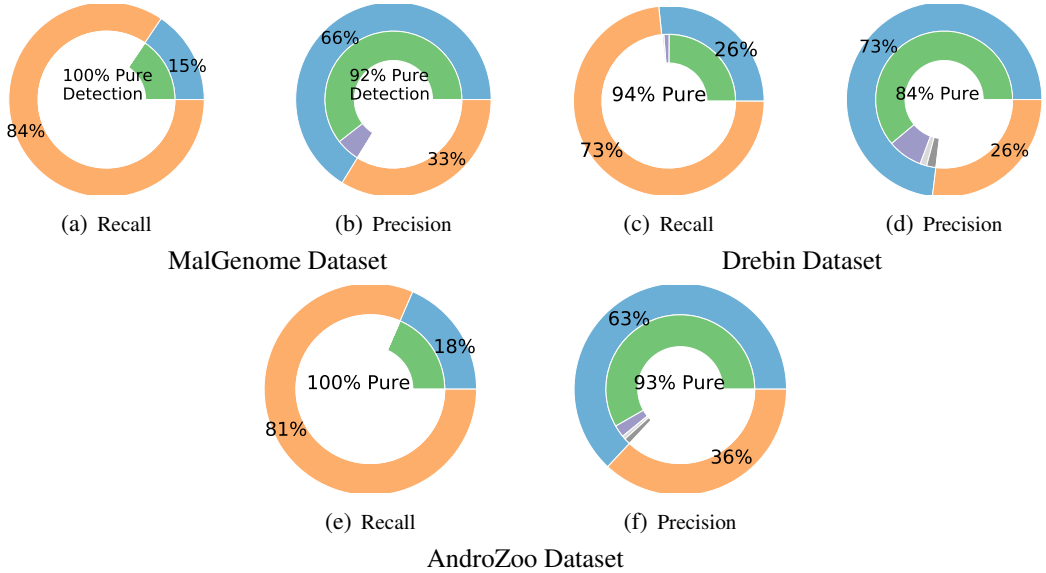


(a) Similairty and Content (b) Similarity Threshold (c) Content Threshold

Figure 3.24: Community Detection Time Analysis on AndroZoo

3.5 Case Study: Recall and Precision Settings

In this section, we evaluate Cypider framework with respect to recall and precision. We aim to assess Cypider performance in terms of *purity* and *coverage* in case the security practitioner focuses on having: (i) maximum recall (a minimum false detection), or (ii) maximum precision (maximum coverage). Both recall and precision settings are common in real deployments. We tune the recall and precision settings by adjusting Cypider hyper-parameters to reach the set goal.



(a) Recall (b) Precision (c) Recall (d) Precision

MalGenome Dataset Drebin Dataset

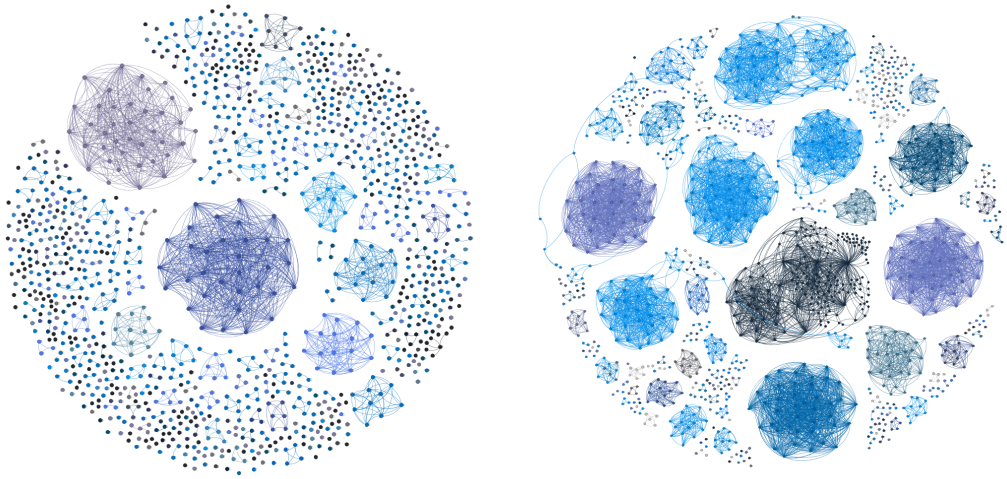
(e) Recall (f) Precision

AndroZoo Dataset

Figure 3.25: Performance under Recall/Precision Settings

Figure 3.25 presents Cypider malware only performance on different datasets (*MalGenome, Drebin, AndroZoo*) under recall and precision settings. In the recall setting, Cypider achieves 95%

to 100% purity while maintaining 15% to 26% malware coverage. Therefore, we detect about 20% (on average) of the input malware in form of communities with 98% purity (Figure 3.25, recall charts). On the other hand, Cypider achieves 63% to 73% malware coverage while maintaining 84% to 93% purity, as shown in Figure 3.25 (precision charts).



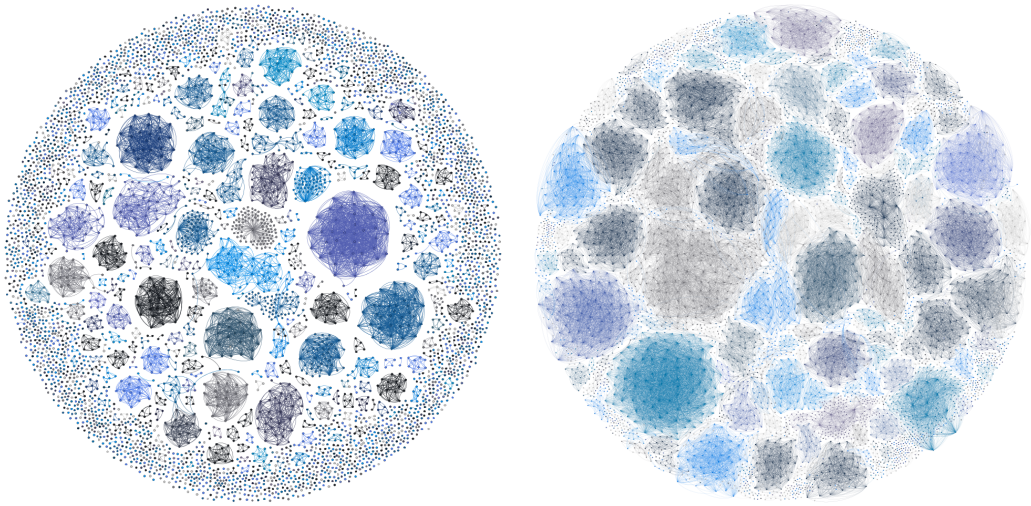
(a) Recall Settings

(b) Precision Settings

Figure 3.26: Malgenome Similarity Networks

The contrast between the recall and precision settings is more clearly visible in the similarity network, as shown in Figures 3.26 and 3.27 for *MalGenome* and *Drebin* datasets respectively. The aforementioned figures present each malware family in a different color. Malware communities depicted with more than one color contain more than one malware family. Pure malware communities have only one color in the edges and nodes. We notice more detected malware communities in the similarity network in the precision settings. In contrast, in the recall similarity network, we notice fewer malware communities, and most of the nodes are part of any community (not detected).

Figure 3.28 depicts Cypider mixed performance under recall and precision settings for *MalGenome*, *Drebin* and *AndroZoo* datasets. The most noticeable result is that all the detected benign communities have perfect purity metrics under both recall and precision settings. Moreover, benign coverage is less than the malware coverage under all settings. In other words, Cypider could bring benign samples during clustering but gathered in pure communities, which is very helpful in case of manual



(a) Recall Settings (b) Precision Settings

Figure 3.27: Drebin Similarity Networks

investigations.

The difference between recall and precision settings in the mixed scenario is more clearly visible in the similarity networks. Figures 3.29 and 3.30 show the recall and precision similarity network of *MalGenome* and *Drebin* datasets respectively. Darker color communities contain malware samples, and lighter color communities contain benign samples. We notice a clear separation between malicious communities and benign ones. Also, more numerous and larger communities have been detected under the precision setting compared to the recall setting.

Tables 3.12 and 3.13 detail Cypider performance under the recall and the precision settings in terms of coverage/purity and number of detected/pure communities respectively.

3.6 Case Study: Obfuscation

In this section, we investigate the robustness of Cypider framework against common obfuscation techniques and code transformation in general. We employ *PRAGuard* obfuscated Android malware, which contains $11k$ samples, along with benign samples from *AndroZoo* dataset. Table 3.14 details Cypider performance on the malware and the mixed scenarios. We compare Cypider performance before and after applying a single or a combination of obfuscation techniques, as

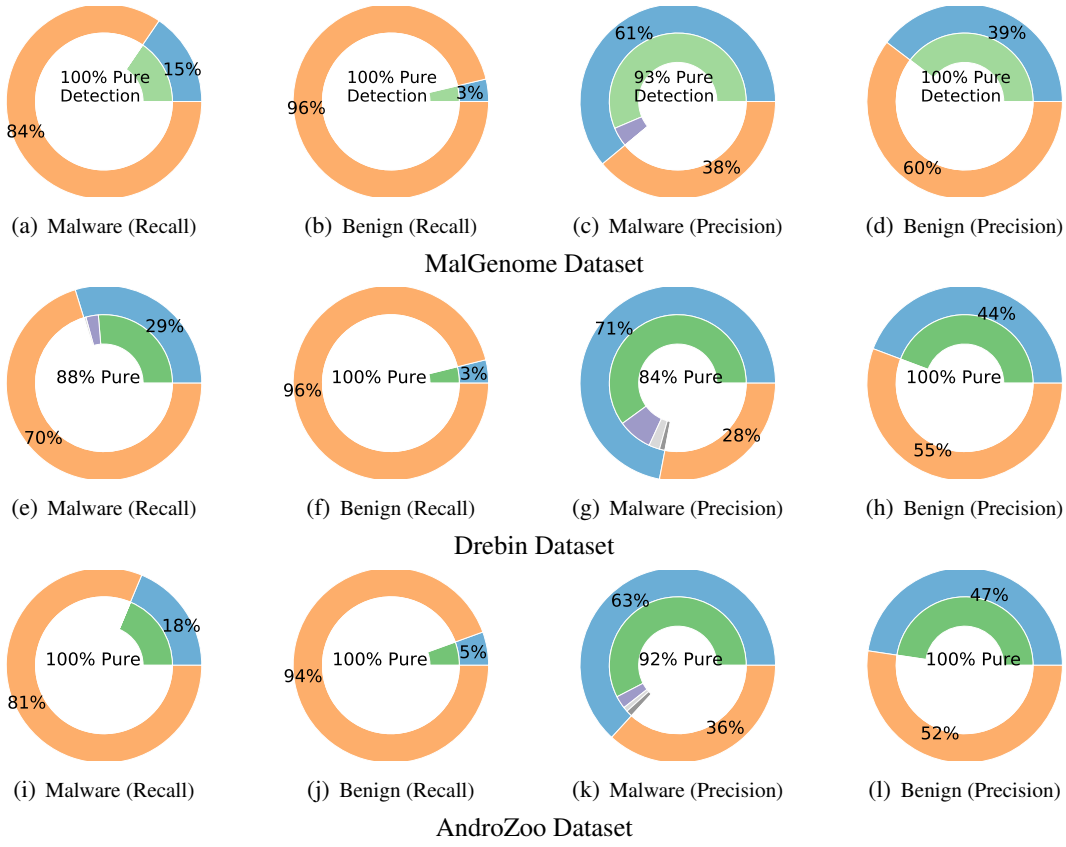
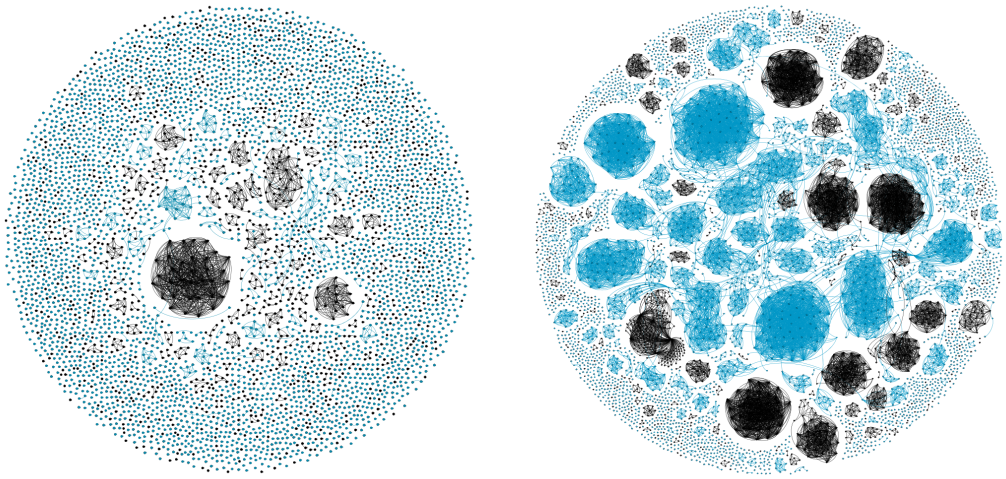


Figure 3.28: Performance under Mixed Recall/Precision Settings

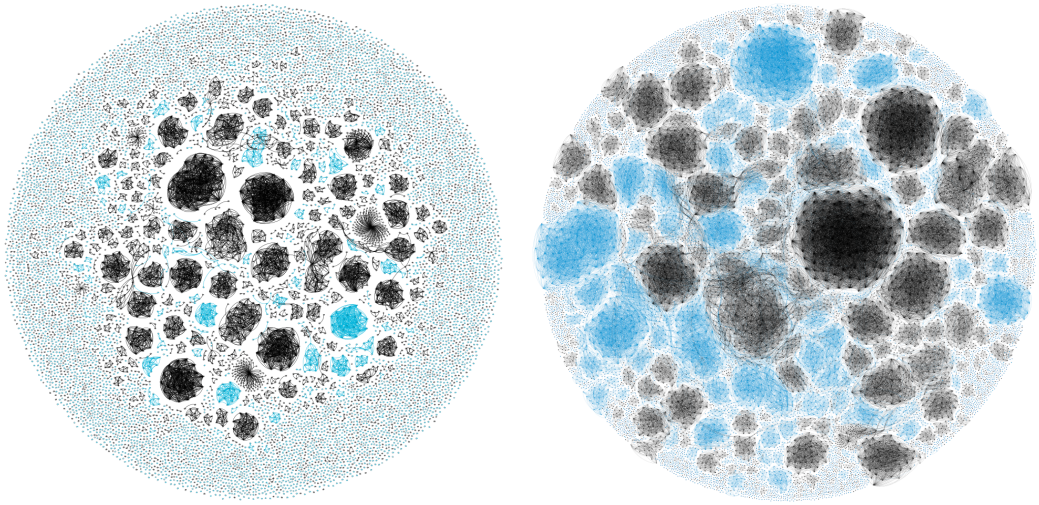
shown in Table 3.14.

The evaluation results show that common obfuscation techniques have a limited effect on Cypider performance in general ($60-77\%$ coverage and $71-99\%$ purity). *Class encryption* obfuscation decreases the coverage from 66% in the non-obfuscated dataset to $36-38\%$. However, *Class encryption* does not affect the purity. Similarly, *Reflection* obfuscation technique dropped down the purity to $67-71\%$ compared to the original dataset but does not affect the coverage performance. To strengthen our findings (Table 3.14) on *PRAGuard* obfuscation dataset, we build our obfuscation dataset using *DroidChameleon* obfuscation tool. We obfuscate *Drebin* malware dataset ($5k$ malware samples) and benign samples from *AndroZoo* dataset ($5k$ malware samples). The result is $100k$ samples ($50k$ malware and $50k$ benign) from different obfuscation settings, as shown in Table 3.15. Similar to *PRAGuard* experiment, we compare Cypider performance before obfuscation (original *Drebin* dataset) and after obfuscation. However, this experiment is different from the



(a) Recall Settings (b) Precision Settings

Figure 3.29: Malgenome Mixed Similarity Network



(a) Recall Settings (b) Precision Settings

Figure 3.30: Drebin Mixed Similarity Network

PRAGuard one because both benign and malicious samples are obfuscated.

Table 3.15 details the result of Cypider framework on the different obfuscation techniques. The most noticeable is that the obfuscation techniques provided by the *DroidChameleon* tool have a limited effect on the clustering clustering. All the performance metrics are remain stable on both

Evaluation Setup			Dataset Size			Coverage/Purity %		
Dataset	**Scenario**	**Settings**	**#Benign**	**#Malware**	**#Total**			
Malgenome	**Malware**	*Recall*	/	1.23k	1.23k	/	15.56%/99.0%	15.56%/99.0%
		Precision	/	1.23k	1.23k	/	66.21%/91.0%	66.21%/91.0%
	Mixed	*Recall*	4.03k	1.23k	5.26k	3.7%/100.0%	15.48%/99.0%	6.46%/100.0%
		Precision	4.03k	1.23k	5.26k	39.66%/100.0%	61.02%/92.0%	44.68%/98.0%
Drebin	**Malware**	*Recall*	/	5.55k	5.55k	/	26.69%/93.0%	26.69%/93.0%
		Precision	/	5.55k	5.55k	/	73.08%/84.0%	73.08%/84.0%
	Mixed	*Recall*	7.64k	5.55k	13.19k	3.85%/100.0%	29.77%/88.0%	14.74%/90.0%
		Precision	7.64k	5.55k	13.19k	44.27%/100.0%	72.0%/83.0%	55.93%/91.0%
AndroZoo	**Malware**	*Recall*	/	66.76k	66.76k	0.0%/0.0%	18.48%/100.0%	18.48%/100.0%
		Precision	/	66.76k	66.76k	0.0%/0.0%	63.03%/92.0%	63.03%/92.0%
	Mixed	*Recall*	44.18k	66.76k	110.94k	5.52%/100.0%	18.7%/100.0%	13.45%/100.0%
		Precision	44.18k	66.76k	110.94k	47.63%/100.0%	63.19%/91.0%	56.99%/94.0%

Table 3.12: Coverage and Purity Details

Evaluation Setup			Dataset Size			#Communities/#Pure		
Dataset	**Scenario**	**Settings**	**#Benign**	**#Malware**	**#Total**	**Bengin**	**Malware**	**Overall**
Malgenome	**Malware**	*Recall*	/	1.23k	1.23k	/	15/15	15/15
		Precision	/	1.23k	1.23k	/	35/31	35/31
	Mixed	*Recall*	4.03k	1.23k	5.26k	18/18	17/17	35/35
		Precision	4.03k	1.23k	5.26k	71/71	34/31	105/102
Drebin	**Malware**	*Recall*	/	5.55k	5.55k	/	95/89	95/89
		Precision	/	5.55k	5.55k	/	155/136	155/136
	Mixed	*Recall*	7.64k	5.55k	13.19k	30/30	109/102	139/132
		Precision	7.64k	5.55k	13.19k	125/125	152/132	277/257
AndroZoo	**Malware**	*Recall*	/	66.76k	66.76k	/	800/798	800/798
		Precision	/	66.76k	66.76k	/	1355/1291	1355/1291
	Mixed	*Recall*	44.18k	66.76k	110.94k	176/176	828/826	1004/1002
		Precision	44.18k	66.76k	110.94k	586/586	1321/1250	1907/1836

Table 3.13: Number of Detected/Pure Communities Details

Evaluation Setup		Coverage/Purity %			#Communities/#Pure		
Scenario	**Obfuscation**	**Bengin**	**Malware**	**Overall**	**Bengin**	**Malware**	**Overall**
Malware	**Malgenome (Orignal)**	/	66.2%/92.3%	66.2%/92.3%	/	35/31	17/17
	(1) TRIVIAL	/	60.3%/99.8%	60.3%/99.8%	/	34/34	35/31
	(2) STRING ENCRYPTION	/	63.5%/96.7%	63.5%/96.7%	/	34/32	35/31
	(3) REFLECTION	/	70.8%/71.9%	70.8%/71.9%	/	30/27	35/31
	(4) CLASS ENCRYPTION	/	38.3%/98.5%	38.3%/98.5%	/	27/26	35/31
	(1) & (2)	/	52.4%/99.8%	52.4%/99.8%	/	42/42	35/31
	(1) & (2) & (3)	/	65.1%/67.5%	65.1%/67.5%	/	38/34	35/31
	(1) & (2) & (3) & (4)	/	36.3%/99.7%	36.3%/99.7%	/	39/39	35/31
Mixed	**Malgenome (Orignal)**	52.84%/100.0%	45.44%/95.0%	51.1%/99.0%	63/63	36/32	99/95
	(1) TRIVIAL	50.98%/100.0%	65.72%/90.0%	54.3%/97.0%	66/66	38/32	104/98
	(2) STRING ENCRYPTION	52.47%/100.0%	68.41%/93.0%	56.06%/98.0%	66/66	33/30	99/96
	(3) REFLECTION	51.45%/100.0%	77.23%/63.0%	57.21%/89.0%	65/65	29/21	94/86
	(4) CLASS ENCRYPTION	52.84%/100.0%	45.44%/95.0%	51.1%/99.0%	63/63	36/32	99/95
	(1) & (2)	/	/	/	/	/	/
	(1) & (2) & (3)	/	/	/	/	/	/
	(1) & (2) & (3) & (4)	49.04%/100.0%	44.64%/94.0%	48.01%/99.0%	66/66	39/37	105/103

Table 3.14: Performance on Obfuscated - PRAGaurd Dataset

Scenario	Evaluation Setup Obfuscation	Coverage/Purity % Bengin	 Malware	 Overall	#Communities/#Pure Bengin	 Malware	 Overall
Mixed	**Drebin (Original)**	44.27%/100.0%	72.0%/83.0%	55.93%/91.0%	125/125	152/132	277/257
	Class Renaming	41.71%/100.0%	72.61%/83.0%	54.64%/91.0%	129/129	156/135	285/264
	Method Renaming	42.02%/100.0%	71.08%/83.0%	54.19%/91.0%	121/121	149/129	270/250
	Field Renaming	43.59%/100.0%	72.01%/83.0%	55.49%/91.0%	128/128	148/127	276/255
	Code Reordering	43.43%/100.0%	71.49%/83.0%	55.19%/91.0%	127/127	155/135	282/262
	Debug Information Removing	44.34%/100.0%	72.09%/83.0%	55.96%/91.0%	117/117	151/130	268/247
	Junk Code Insertion	40.71%/100.0%	71.81%/83.0%	53.73%/90.0%	124/124	153/132	277/256
	Instruction Insertion	42.65%/100.0%	71.25%/83.0%	54.63%/91.0%	120/120	156/136	276/256
	String Encryption	43.2%/100.0%	72.16%/83.0%	55.32%/91.0%	133/133	147/127	280/260
	Array Encryption	43.55%/100.0%	71.93%/83.0%	55.42%/91.0%	125/125	152/131	277/256
Malware	**Drebin (Original)**	/	73.08%/84.0%	73.08%/84.0%	/	155/136	155/136
	Class Renaming	/	74.14%/84.0%	74.14%/84.0%	/	160/137	160/137
	Method Renaming	/	72.66%/83.0%	72.66%/83.0%	/	159/136	159/136
	Field Renaming	/	73.75%/83.0%	73.75%/83.0%	/	155/132	155/132
	Code Reordering	/	74.07%/83.0%	74.07%/83.0%	/	158/135	158/135
	Debug Information Removing	/	72.92%/83.0%	72.92%/83.0%	/	155/132	155/132
	Junk Code Insertion	/	73.86%/83.0%	73.86%/83.0%	/	157/135	157/135
	Instruction Insertion	/	73.96%/85.0%	73.96%/85.0%	/	160/137	160/137
	String Encryption	/	73.8%/83.0%	73.8%/83.0%	/	155/132	155/132
	Array Encryption	/	73.8%/83.0%	73.8%/83.0%	/	155/133	155/133

Table 3.15: Performance on Obfuscation - Drebin Dataset

non-obfuscated and obfuscated samples under the malware and mixed scenarios. We argue that Cypider framework is resilient to common obfuscation and code transformation techniques because our framework considers many APK contents for feature extraction. Therefore, the obfuscation techniques can affect one APK content, but Cypider is able to leverage other contents to fingerprint malware sample and compute the similarity with other malware samples.

3.7 Case Study: Win32 Malware

In this section, we study the application of Cypider on different platform malware. Specifically, we employ Cypider to group Win32 malware samples into communities of the same malware family. Our goal is to check the effectiveness of Cypider outside the Android world.

3.7.1 Dataset Description

A Win32 malware dataset is the first thing that we looked for to carry out this case study. We looked for a large, freely available labeled malware dataset that contains several families. All those criteria are met in Microsoft malware dataset[3] that has been used in a Kaggle[4] competition in 2015.

[3]https://www.kaggle.com/c/microsoft-malware-prediction
[4]https://kaggle.com

	Family	# Sample
1	Kelihos3	2942
2	Lollipop	2478
3	Ramnit	1541
4	Obfuscator	1228
5	Gatak	1013
6	Tracur	751
7	Vundo	475
8	Kelihos1	398
9	Simda	42
	Total	10868

Table 3.16: Microsoft Kaggle Competition Malware Dataset

The dataset contains about $11k$ ($\approx$ 500GB) malware samples from nine Win32 malware families, as shown in Table 3.16. The Dataset contains for each malware the actual binary and its Intel x86 assembly in separate files.

3.7.2 Static Features

Win32 malware has a different binary compared to Android packages in terms of structure and assembly. Therefore, we need to re-engineer new static features to compute Cypider features vectors. For the sake of this case study, we used only three content features, namely: Intel x86 assembly opcodes, function calls, and binary bytes. Further features vectors for x86 binary samples could be investigated as part of future work. Opcode or operation code is the first part of the machine instruction to be executed on the targeted machine. Using opcodes in malware fingerprinting tends to provide more resiliency to some obfuscation techniques that affect other portions of machine instructions. We parse assembly files to extract the opcodes, in which we preserve the order of appearance in a file. In our context, a function call is any system call that requests a service/resource from the underlying operating system using know APIs. We parse the assembly file to extract ordered function calls. Binary bytes are the contiguous bytes of a sample binary representation. We obtain a byte sequence from the binary content of a malware sample. For all of the previous content features, we use N-grams (4-grams) and feature hashing techniques (size=2^{16}). Afterward, we apply principal component analysis (PCA) for dimensionality reduction to produce the final compact content vectors (size=100).

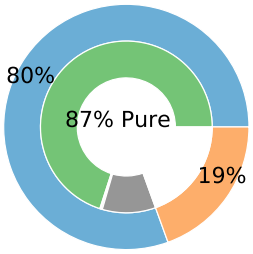


Figure 3.31: Framework Performance on Win32 Malware

Evaluation Setup	Dataset Size	#Communities/#Pure	Coverage/Purity %
Microsoft Kaggle Malware	10.87k	160/154	80.52%/87.0%

Table 3.17: Purity and the Detected Communities on Win32

3.7.3 Findings

Figure 3.31 depicts Cypider performance on Microsoft malware competition dataset. Even though we used only three features content, Cypider still achieves high performance in terms of community purity and the overall coverage. Cypider is able to cluster 80% of the dataset with 87% purity. Besides, most of the impure communities (13%) contains samples from only two or three malware families.

Figure 3.32 presents the result of applying Cypider before filtering unassociated malware samples. Each color scale in Figure 3.32 represents a malware family in the dataset (Table 3.16). Figure 3.32 reveals different observations in Cypider similarity network, as follows. (1) The detected malware communities have a highly connected network. (2) Some malware communities are composed of the connection of multiple sub malware communities networks. The sub-communities could represent malware variants of the main malware family.

Table 3.17 summarizes the performance result of Cypider on Win32 malware dataset. Cypider is able to group with high purity about $9k$ samples into 160 malicious communities. Specifically, it identifies 154 pure communities out of 160 detected communities.

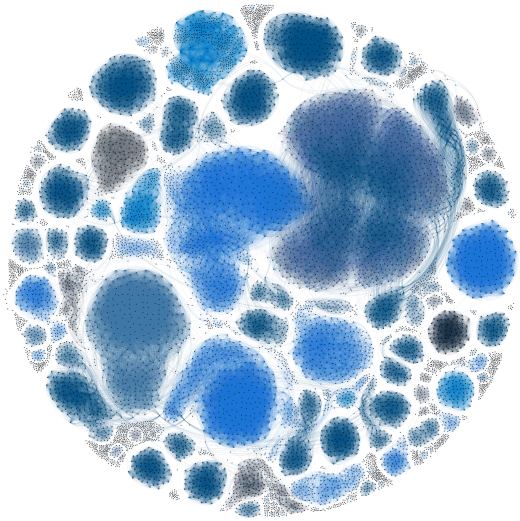


Figure 3.32: Similarity Network of Win32 Malware Dataset

3.8 Summary

In this chapter, we have presented a fuzzy fingerprinting approach for investigating Android malware variations. To this end, we have proposed APK-DNA, a novel fingerprint that captures not only the binary of the *APK* file but also both its semantics. This is of paramount importance since it allows the generated fingerprints to be highly resistant to app changes, which is a significant advantage compared to traditional fuzzy hashing techniques. We have leveraged the proposed APK-DNA fingerprint to design and implement an innovative, efficient and scalable framework for Android malware detection, called Cypider. In essence, the detection mechanism relies on the community concept. Cypider provides a systematic framework that can generate a fingerprint for each community, enabling the identification of known and unknown malicious communities. Cypider has been implemented and evaluated on different malicious and mixed datasets. Our findings show that Cypider is a valuable and promising framework for the detection of malicious communities. Cypider only needs few seconds to build a network similarity of a large number of apps. The community fingerprinting results are very promising as 87% of the detection is achieved.

In the next chapter, we present another fuzzy fingerprinting approach for Android malware detection with the following specificities: (i) We rely on dynamic analysis instead of static analysis.

(ii) We build on top the dynamic analysis fingerprints a classification system using supervised machine learning instead of the clustering approach employed in this chapter.

Chapter 4

Android Malware Fingerprinting Using Dynamic Analysis

4.1 Overview

In this chapter, we elaborate a data driven framework for detecting Android malware using automatically engineered features derived from dynamic analyses. The state-of-the-art solutions, such as [91, 134, 171], rely on manual feature engineering in malware detection. For example, StormDroid [91] uses *Sendsms* and *Recvnet* dynamic features, which are chosen based on statistical analysis, for Android malware detection. As another example, the authors in [141] used specific features to build behavioral graphs for Win32 malware detection. The security features may change based on the execution environment despite the target platform. For instance, the authors in [91] and in [70] used different security features due to the difference between the execution environments. In the context of a security application, we are looking for a portable framework for malware detection based on the behavioral reports across a variety of platforms, architectures, and execution environments. The security analyst would be able to rely on this plug-and-play framework with a minimum effort in terms of feature engineerning. We plug the behavioral analysis reports for the training. Afterward, we employ the produced classification model on new reports without an explicit security feature engineering as in [89, 91, 141]. This previous process works virtually on any behavioral reports.

First, we propose a novel fingerprinting approach, namely DySign, which aims at generating

a signature that is based on the dynamic analysis of Android malware apps. Second, relying on DySign fingerprinting, we propose, MalDy, a portable framework for malware detection and family threat investigation based on behavioral reports. MalDy framework is built on top of Natural Language Processing (NLP) modeling and supervised machine learning techniques. The main idea is to formalize a behavioral report, agnostic to the execution environment, into a Bag of Words (BoW) where the features are the reports' words. Afterward, we leverage machine learning techniques to automatically discover relevant security features that help differentiate and attribute malware. The result is MalDy, a portable (Section 4.3.4), effective (Section 4.3.4), and efficient (Section 4.3.4) framework for malware analysis.

4.1.1 Threat Model

We position MalDy as a generic malware analysis tool. MalDy considers only behavioral reports generated from the execution of program binary inside a sand-boxing environment. Therefore, MalDy is by design resilient to binary code static analysis transformation like packing, compression, and dynamic loading. MalDy performance depends on the quality of the collected reports. The more security information and features are provided about malware samples in the reports, the more accurate MalDy could differentiate malware from benign and attribute to known families. The malware execution time and the random event generator of the sandboxing may have a considerable impact on MalDy because they affect the quality of the behavioral reports. Anti-emulation techniques, used to evade dynamic analysis, could be challenging for MalDy framework. However, this issue is related to the choice of the underlying execution environment.

4.2 Dynamic Analysis Fingerprints

The main aim of DySign is to generate an approximate fingerprint from dynamic analysis output of malicious apps. Fingerprints are generated for each app in a database of known apps. Our primary concern, after accuracy, is the scalability in the fingerprinting of existing known malware and matching the generated fingerprints against new apps to check possible maliciousness. DySign is intended to be the first fingerprint defense line, along with static file fuzzy fingerprinting, to tackle

the overwhelming volume of malicious apps encountered daily. DySign has two primary usage scenarios: (i) *Mobile apps monitoring*: In this scenario, we have a set of installed apps that run in a given smart device. Having a runtime report database of these apps would help DySign to periodically fingerprint the behaviors of these apps to check for the existence of abnormal behaviors. In this scenario, DySign could raise an alert of behavior change after a suspicious update or a hack; (ii) *Cloud service analyzer*: In this scenario, DySign is used as a core of cloud checking service of the received analysis reports (either automatically using a collection service inside devices or manually by users' submissions) from Android device of suspicious apps. The goal is to match runtime analysis against malicious apps. These scenarios are general applications of DySign. However, we believe that it can be extended to many other usages due to the simplicity and scalability of DySign.

4.2.1 Methodology

In this section, we present the architecture of DySign, Figure 6.1 along with the different phases of the proposed system.

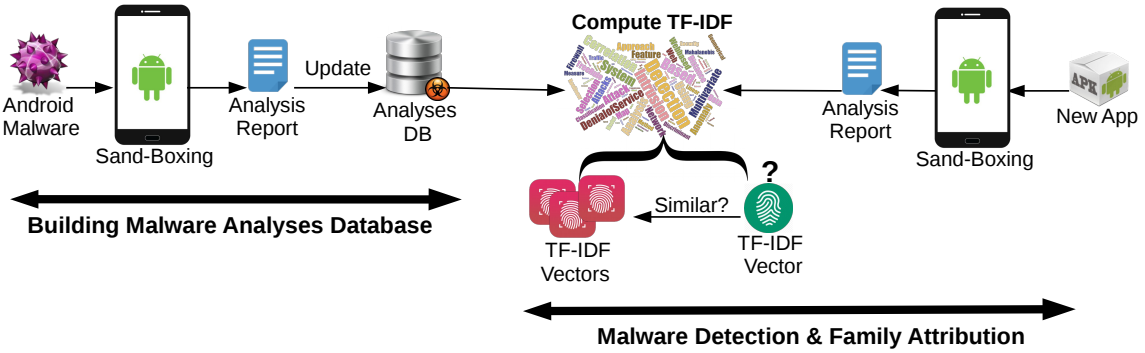


Figure 4.1: DySign Approach

Fingerprint Computation

Our ultimate goal is to automatically fingerprint Android malware based on dynamic analysis. To this end, we use *natural language processing* techniques, where we consider the output of the dynamic analysis as a plain text file and model it as a *bag of words*. The latter treats the text document as a set of words separated by predefined delimiters such as *spaces* and *curly brackets*. Given a set of analysis reports as a *bag of words*, we compute a *relative fingerprint* for each report based on

the word frequency. More specifically, we segregate the reports by giving high weight to the words with a high frequency (number of occurrence) in a given report and low frequency in the others. The result is a vector of words' weights for each analysis report. To compute DySign's vector, we leverage the so-called *Term Frequency-Inverse Document Frequency* (*tf-idf*) [190], a well-known technique adopted in the fields of *information retrieval* and *natural language processing*. The latter computes vectors from input text documents by considering both frequency in individual documents and the whole set. Let $D = \{d_1, d_2, \ldots, d_n\}$ be a set of text documents, where n is the number of documents, and let $d = \{w_1, w_2, \ldots, w_m\}$ be a document, where m is the number of words in d. The *tf-idf* of a word w and document d is the product of *term frequency* of w in d and the *inverse document frequency* of w, as shown in Equation 4. The *term frequency* (Equation 5) is the occurrence number of w in d. Finally, the *inverse document frequency* of w (Equation 6) represents the lograthm of the ratio between the number of documents n and the number of documents that contain w plus one. The computation of *tf-idf* is very scalable, which suites our needs.

$$\textit{tf-idf}(w, d) = \textit{tf}(w, d) \times \textit{idf}(w) \tag{4}$$

$$\textit{tf}(w, d) = |\{w \in d,\ d = \{w_1, w_2, ...w_n\} : w = w_i\}| \tag{5}$$

$$\textit{idf}(w) = log\frac{|D|}{1 + |\{d : w \in d\}|} \tag{6}$$

The result of *tf-idf* is a set of vectors $V = \{v_1, v_2, \ldots, v_n\}$ (DySign fingerprints) of word weights for each document $d \in D$. Computing the similarity using *DySign* is straightforward using the *cosine similarity* as shown in Equation 7.

$$\text{cosine-similarity}(v_1, v_2) = \cos(\theta) = \frac{v_1 \cdot v_2}{||v_1||||v_2||} \tag{7}$$

We use an illustrating example to answer a crucial question, *How can DySign be used for Android malware detection and family attribution?*. Thus, we compute DySign fingerprints from the analysis reports of malware samples from Drebin malware dataset [73, 175] along with benign

apps downloaded from Google Play [28]. The example is summarized in Tables 4.1 and 4.2. How DySign is used for malware detection is illustrated in Table 4.1. This example shows the potential of DySign in distinguishing between malware and benign apps.

#	*App1*	*App2*	*TFIDF Cosine*
1	00453ca8 (FakeInst)[1]	com.BigBawb.coin.apk	0.19
2	00453ca8 (FakeInst)	com.interestcalculator.apk	0.21
3	21262a59 (FakeInst)	com.sleggi.MiFreetime.apk	0.16
4	00453ca8 (FakeInst)	21262a59 (FakeInst)	0.42
5	00453ca8 (FakeInst)	com.sleggi.MiFreetime.apk	0.27

[1] First 8 characters from malware hash and its malware family.

Table 4.1: DySign and Android Malware Detection

As shown in Table 4.2, DySign could be used to segregate between Android malware families by requiring a higher similarity between fingerprints of the same malware family. Based on these insights, we generalize and build a system on top of DySign for Android malware detection and family attribution.

#	*Malware1*	*Malware2*	*TFIDF Cosine*
1	090b5be2 (Plankton)[2]	bedf51a5 (DroidKungFu)	0.56
2	149bde78 (Plankton)	bedf51a5 (DroidKungFu)	0.46
3	090b5be2 (Plankton)	149bde78 (Plankton)	0.71

[2] First 8 characters from malware hash and its malware family.

Table 4.2: DySign and Android Malware Family Attribution

How DySign is agnostic to malware samples and families? DySign is agnostic to malware samples and families by design since no features are explicitly extracted for a given malware family or a sample. In other words, DySign considers an analysis report as a bag of words. It only finds the frequency of the word in a document relative to the other ones. This ensures that the extracted DySign information is broad enough to cover most malware samples without relying on malware specific features.

Architecture Overview

In this section, we present the architecture of DySign framework for Android malware detection. There are two main processes in DySign framework. i) The first process is building the analysis report database. The initial phase of this process consists of sandboxing and inserting the reports into the database of known Android malware (Algorithm 5). Afterward, the process proceeds as a

continuous task of updating the report's database with new apps (Algorithm 6).

Algorithm 5: First Setup of Analysis Report Database

```
Input : MalDataset: APK Files of Known Malware
        BenDataset: APK Files of Some Benign Apps
begin
    foreach Apk ∈ MalDataset do
        Report ← SandBoxing(Apk);
        WordBag ← getWordBag(Report);
        SaveDatabase(WordBag);
    end
    foreach Apk ∈ BenDataset do
        Report ← SandBoxing(Apk);
        WordBag ← getWordBag(Report);
        SaveDatabase(WordBag);
    end
    LunchUpdateProcess()
end
```

Algorithm 6: Updating Analysis Report Database

```
Input : NewUpdateApp: Update App File (APK)
begin
    while True do
        if ∃ NewUpdateApp then
            NewReport ← SandBoxing(NewUpdateApp);
            WordBag ← getWordBag(NewReport);
            SaveDatabase(WordBag);
        end
    end
end
```

ii) The second process is the detection, in which we check the runtime behaviors of newly received apps against known malware behaviors. First, the new app is executed in a sandboxing environment during a time T to get the analysis report. The latter will be used along with the database reports to compute the DySign fingerprint using *tf-idf*. Finally, we compute the similarity between the DySign fingerprint of the new app and the existing fingerprints to identify whether it is malicious or not and its family in the case that the app is identified as malicious. The complete DySign process is presented in Algorithm 7. Using DySing's fingerprint, we do not only detect malware but also attribute the unknown samples to their Android malware families. Further, we can also ascribe a family to the unknown samples if we already have samples of this family in DySign's dynamic analysis database. Algorithm 7 describes the process of generating a dynamic fingerprint.

A cornerstone in DySign framework is the sandboxing system, which heavily influences the

Algorithm 7: DySign Framework Detection Process

Input : $Database$: Analysis Reports Of DySign Database
$NewApp$: New App File (APK)
Output: $Decision$: {Bengin or Malicious}
$Family$: Android Malware Family

```
begin
    NewReport ← SandBoxing(NewApp);
    dbVectors, NewVector ← TFIDF(dbReports, NewReport);
    MaxSim ← 0;
    Decision ← Benign;
    Family ← ∅;
    foreach Vec ∈ dbVectors do
        Sim ← Similarity(Vec, NewVector);
        if Sim > MaxSim then
            MaxSim ← Sim;
            Decision ← getDecision(Vec);
            Family ← getAndroidFamily(Vec);
        end
    end
    return Decision, Family;
end
```

produced analysis reports. We use *DroidBox* [47], a well-established sandboxing environment based on the Android software emulator [12] provided by Google Android SDK [33]. Running the app may not lead to a sufficient coverage of the executed app. As such, to simulate the user interaction with the apps, we leverage *MonkeyRunner* [48], which produces random UI actions aiming for a broader execution coverage. Also, the similarity computation could be a bottleneck for DySign and could lead to inefficient matching against new unknown apps. To address this issue, we resort to LSH K-Nearest Neighbor (KNN) [80]. The performance of similarity computation needs to be much faster than the brute-force computation. To this end, we leverage *Locality Sensitive Hashing* (LSH) techniques, and more precisely *LSH Forest* [80], a tunable high-performance algorithm for similarity computation as described in Chapter 3 (Section 3.3.4).

4.2.2 Experimental Results

In this section, we present the evaluation results of our proposed system. The implementation subsection shows the setup of our experiments. To evaluate the performance of malware detection using DySign, we use a mixed dataset, i.e., malware and benign apps. As for the evaluation of the attribution performance, we use a malware-only dataset.

Evaluation Dataset

The first step towards evaluating DySign is to select appropriate datasets that can be utilized for Android malware fingerprinting. Obtaining representative datasets is a fundamental challenge, and there is certainly a strong need for reference datasets. Hence, the utilized dataset consists of: (1) *malware-only* dataset using the well-known Drebin dataset [73, 175], and ii) *mixed* dataset using Drebin dataset along with benign apps downloaded from Google Play [28]. Statistics about the dataset are presented in Table 4.3. In Table 4.3, we use a subset of $3,414$ Android malware samples, from Drebin dataset, distributed across 8 families. To make the data more balanced, we exclude from this dataset all malware families with few samples due to the high skewness of the dataset. This would prevent having, for instance, a family with 800 samples and other families with only 1, 2, or even 20 samples.

	Drebin Dataset	*Drebin Mixed With Benign*
Total Size	3414	8639
Malware	3414	3414
Benign	/	5225

Table 4.3: Android Dataset Description

Results

To evaluate our approach using the previous datasets, we split the training data into ten sets, reserving one set as a testing set and using nine sets as training sets. We use precision (P), recall (R) and F1 metrics, as per Equation 8, where TP, FP, FN represent respectively True Positives, False Positives, and False Negatives.

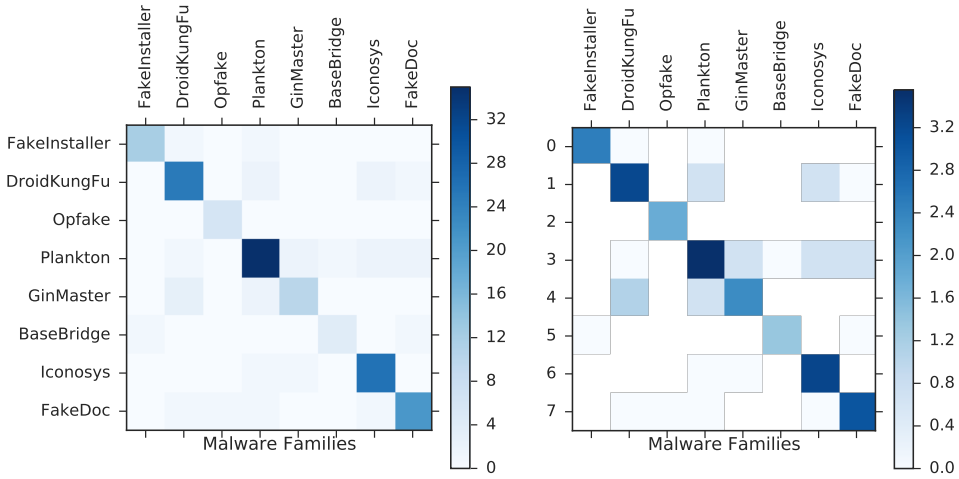
$$P = \frac{TP}{TP + FP},\ R = \frac{TP}{TP + FN},\ F_1 = 2 \times \frac{P \times R}{P + R} \tag{8}$$

Detection Performance: Since the application domain targeted by DySign is much more sensitive to false-positives (FP) than false-negatives (FN), we employ the F-measure, where the results of F_1 measure are summarized in Table 4.4. We use two types of datasets: (i) The mixed dataset, used for detection performance assessment, and (ii) the malware-only dataset, used to assess DySign's family attribution, as presented in Table 4.4. The obtained results show that our approach achieves

good detection and attribution performance in a short time.

	F1-Score	*Precision*	*Recall*	*Time*
Mixed (Detection)	**85%**	**94%**	**78%**	**4min 45s**
Drebin (Attribution)	**80%**	**82%**	**79%**	**2min 20s**

Table 4.4: Detection and Attribution Performance

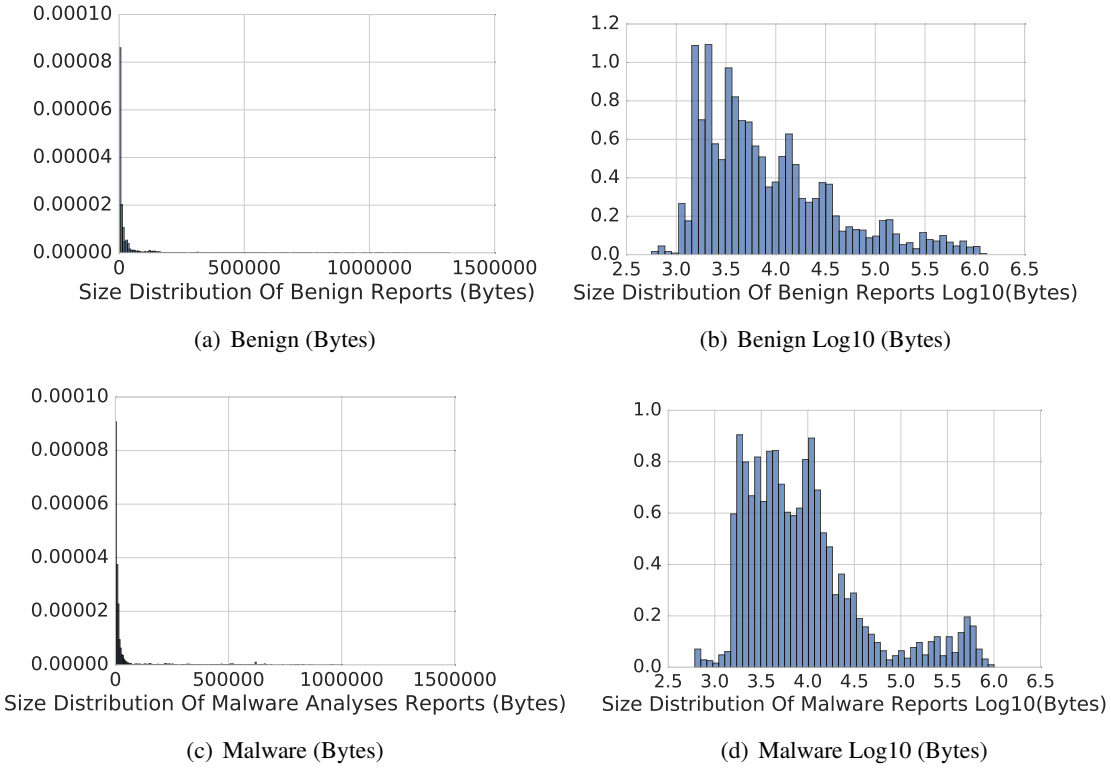


(a) # Malware Samples (b) Log10(#)

Figure 4.2: Family Attribution Evaluation Using Confusion Matrix

Attribution Performance: Figure 4.2 presents the confusion matrix for a more granular view of DySign's family attribution. The darker the diagonal is in the matrix is, the more accurate is the attribution. However, due to unbalanced malware families (Table 5.1), there are some cells in the diagonal that are darker due to the high number of samples in the testing set of that family. For this reason, we apply the log function on the original confusion matrix to have more informative results. Notice that all the produced results are based on the sandbox reports of only $T = 15s$ for each app, whether it is a malware sample or a benign app. Therefore, the accuracy could be significantly improved by having a longer time T.

Reports Size Analysis: Figure 4.3 shows the size distribution of the analysis reports. Figures 4.3(a) and 4.3(c) show the size distribution in *bytes* for benign and malware reports respectively. To enhance the readability of the results, we apply the log function on *byte* distributions. The results are shown in Figures 4.3(b) and 4.3(d) for benign and malware reports. The most noticeable is the



(a) Benign (Bytes)

(b) Benign Log10 (Bytes)

(c) Malware (Bytes)

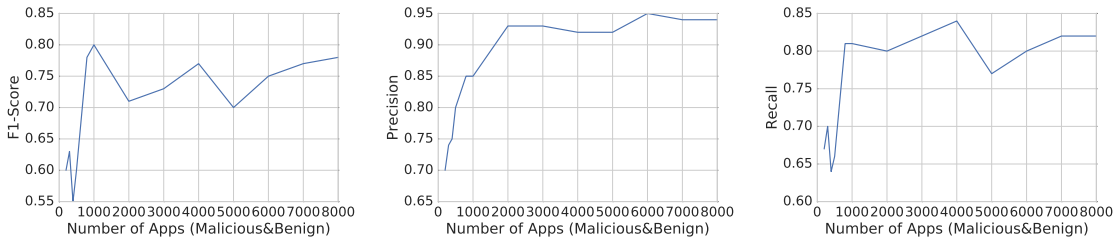
(d) Malware Log10 (Bytes)

Figure 4.3: Sandbox Output Size Distributions

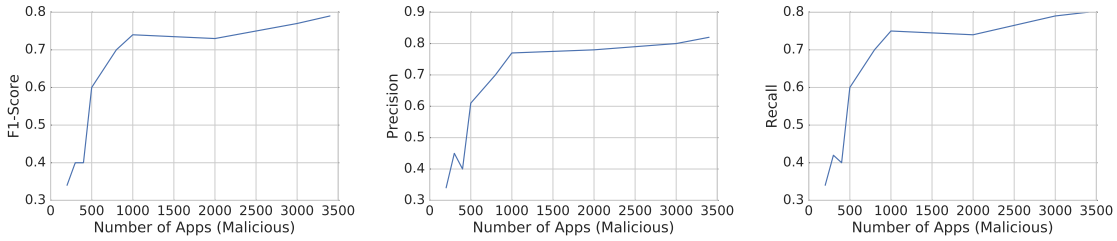
size of the malware compared to the benign reports. Malware reports tend to be bigger than benign ones. This difference happens in a short time since we execute the apps for only $T = 15s$. Our observations show that: (1) Malicious apps tend to have similar behaviors and are generally eager to access the resources to perform their malicious tasks as soon as they are executed. (2) Malware apps tend to be self-driving, i.e., in most cases, they do not need a user interface interaction emulator. Instead, for example, they try to connect to a given IP address with a specific payload.

Accuracy Performance and Dataset Size: Figure 4.4 shows the effect of the dataset size on the detection and family attribution. It also shows the direct relationship between the number of samples in the dataset and accuracy. The more significant is the size of the dataset, the more accurate are the results. However, we could not test for higher scalability since the size of the Drebin dataset limits us after excluding small families. According to the obtained results with our limited dataset, we conclude that by having a bigger dataset, DySign framework could achieve more accurate results.

Scalability Analysis: DySign shows high scalability, as summarized in Figure 4.5. First,

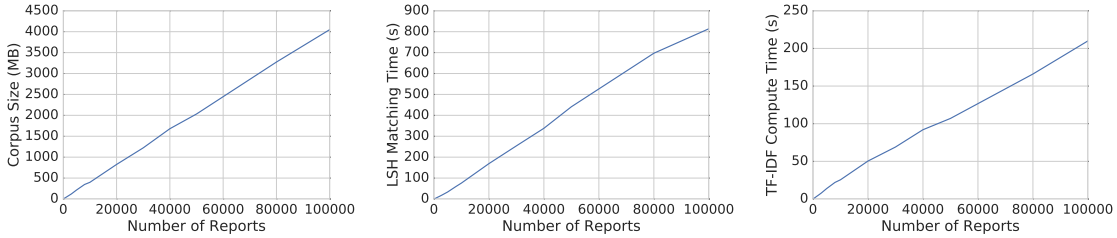


(a) Detection F1-Score (b) Detection Precision (c) Detection Recall



(d) Attribution F1-Score (e) Attribution Precision (f) Attribution Recall

Figure 4.4: Detection Performance over Dataset Size



(a) Corpus Size vs. Number of Reports (b) LSH Matching Time vs Number of Reports (c) TFIDF Computation vs Number of Reports

Figure 4.5: Framework Scalability Analysis

DySign computation time is very fast and linearly scalable with the number of reports. Our system could compute *tf-idf* from $100,000$ analysis reports in about $200s$, as shown in Figure 4.5(c). Notice that we over-sample from our dataset in order to get $100,000$ analysis reports used in the scalability evaluation. Figure 4.5(b) shows the linear growth of *LSH fingerprint matching* with the number of reports. Notice that for a $100,000$-report dataset, we match $10,000$ testing reports against $90,000$ reports in the training dataset.

```
<open_key~key="HKEY_LOCAL_MACHINE\Software\Microsoft\Windows
NT\CurrentVersion\AppCompatFlags\Layers"/> <open_key
key="HKEY_CURRENT_USER\Software\Microsoft\Windows
NT\CurrentVersion\AppCompatFlags\Layers"/> <open_key
key="HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\
LanmanWorkstation\NetworkProvider"/>
</registry_section> <process_section> <enum_processes
apifunction="Process32First" quantity="84"/> <open_process targetpid
   ="308"
desiredaccess="PROCESS_ALL_ACCESS PROCESS_CREATE_PROCESS
   PROCESS_CREATE_THREAD
PROCESS_DUP_HANDLE PROCESS_QUERY_INFORMATION PROCESS_SET_INFORMATION
PROCESS_TERMINATE PROCESS_VM_OPERATION PROCESS_VM_READ PROCESS_VM_WRITE
PROCESS_SET_SESSIONID PROCESS_SET_QUOTA SYNCHRONIZE"
apifunction="NtOpenProcess" successful="1"/>
```

Figure 4.6: Win32 Behavioral Report

4.3 Supervised Malware Detection

In this section, we elaborate a supervised malware detection system on top DySign fingerprint concept.

4.3.1 Overview

The execution of a binary sample (or app) produces textual logs, whether in a controlled environment (software sandbox) or production ones. The execution logs, are composed of a sequence of statements, as the result of the app execution events. Furthermore, each statement is a sequence of words that gives a more granular description of an actual app event. From a security analysis perspective, app behaviors are summarized in an execution report, which is a sequence of statements, and each statement is a sequence of words. Malicious apps tend to have distinguishable behaviors from benign apps, and this difference is translated into words in the behavioral report. Also, similar malicious apps (same malware family) behaviors tend to correspond to related words.

Nowadays, there are many software sandbox solutions for malware investigations. CWSandbox (2006-2011) was one of the first sandbox solutions for production use. It is presently known as ThreatAnalyzer[1], owned by ThreatTrack Security. TheatAnalyzer is a sandbox system for Win32 malware, and it produces behavioral reports that cover most of malware behavioral aspects such as a

[1] https://www.threattrack.com/malware-analysis.aspx

```
"accessedfiles": { "1546331488": "/proc/1006/cmdline","2044518634":
"/data/com.macte.JigsawPuzzle.Romantic/shared_prefs/com.apperhand.global
    .xml",
"296117026":
"/data/com.macte.JigsawPuzzle.Romantic/shared_prefs/com.apperhand.global
    .xml",
"592194838": "/data/data/com.km.installer/shared_prefs/TimeInfo.xml",
"956474991": "/proc/992/cmdline"},"apkName": "fe3a6f2d4c","closenet":
{},"cryptousage": {},"dataleaks": {},"dexclass": { "0.2725639343261719":
     {
    "path": "/data/app/com.km.installer-1.apk", "type": "dexload"}
```


Figure 4.7: Android Behavioral Report

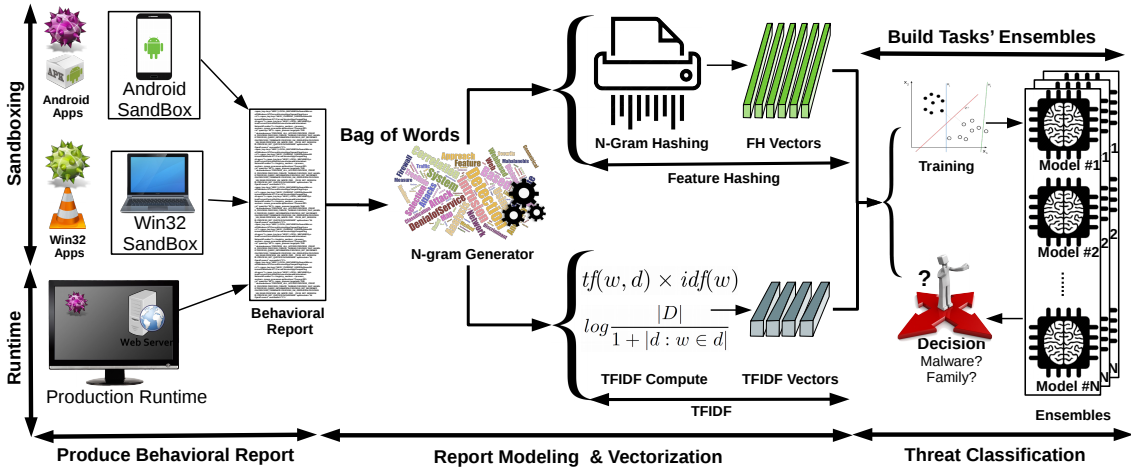


Figure 4.8: MalDy Methodology Overview

file, network, register access records. Figure 4.6 shows a snippet from a behavioral report generated by ThreatAnalyzer. For android malware, we use *DroidBox* [47], a well-established sandbox environment based on Android software emulator [12] provided by Google Android SDK [33]. Figure 4.7 shows a snippet of a behavioral report generated using DroidBox.

Notation

- $X = \{X_{build}, X_{test}\}$: X is the global dataset used to build and report MalDy performance in various tasks. We use the build dataset X_{build} to train and tune the hyper-parameters of MalDy models. The test set X_{test} is used to measure the final performance of MalDy, which is reported in the evaluation section. X is divided randomly and equally to X_{build} (50%) and X_{test} (50%). To build the sub-datasets, we employ the stratified random split on the main

dataset.

- $X_{build} = \{X_{train}, X_{valid}\}$: Build set, X_{build}, is composed of the training set and validation set and used to build MalDy ensembles.

- $m_{build} = m_{train} + m_{valid}$: Build size is the total number of reports used to build MalDy. The training set takes 90% of the build dataset, and the rest is used as a validation set.

- $X_{train} = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{m_{train}}, y_{m_{train}})\}$: The training set, X_{train}, is the training dataset of MalDy machine learning models.

- $m_{train} = |X_{train}|$: The size of m_{train} is the number of reports in the training set.

- $X_{valid} = \{((x_0, y_0), (x_1, y_1), \ldots, (x_{m_{valid}}, y_{m_{valid}})\}$: The validation set, X_{valid}, is the dataset sed to tune the trained model. We choose the hyper-parameters that achieve the best scores on the validation set.

- $m_{valid} = |X_{valid}|$: The size of m_{valid} is the number of reports in the validation set.

- (x_i, y_i) : A single record in X is composed of a single report x_i and its label $y_i \in \{+1, -1\}$. The label meaning depends on the investigation task. In the detection task, a positive values means malware, and a negative means benign. In the family attribution task, a positive means the sample is part of the current model malware family, and a negative means is not.

- $X_{test} = \{((x_0, y_0), (x_1, y_1), .., (x_{m_{test}}, y_{m_{test}})\}$: We use X_{test} to compute and report back the final performance results as presented in the evaluation section (Section 4.3.4).

- $m_{test} = |X_{test}|$: m_{test} is the size the X_{test} and it represents 50% of the global dataset X.

4.3.2 Methodology

In this section, we present the general approach of MalDy, as illustrated in Figure 4.8. The section describes the approach based on the chronological order of the building steps.

Behavioral Reports Generation

MalDy Framework starts from a dataset X of behavioral reports with known labels (malware or benign labels for the detection task, and malware family labels for the attribution task). We consider two primary sources for such reports based on the collection environment. First, we collect the reports from a software sandbox environment [189], in which we execute the binary program, malware, or benign, in a controlled environment (mostly virtual machines). The primary usage of sandboxing in security investigation is to check and analyze the maliciousness of programs. Second, we could collect behavioral reports from a production system in the form of system logs of the running apps. The goal is to investigate the sanity of the apps during their execution. As presented in Section 4.3.1, MalDy employs a word-based approach to model behavioral reports.

Report Vectorization

In this section, we answer the question: how can we model the words in the behavioral report to fit in our classification component? Previous solutions [91, 164] select specific features from the behavioral reports by: (i) extracting relevant security features (ii) manually inspecting and selecting from these features [91]. This process requires manual intervention of the security analyst. Also, it is not scalable since he/she needs to redo this process manually for each new type of behavioral report. In contrast, we are looking for features (words in our case) representation that allows for an automatic feature engineering without the intervention of a security expert.

Build Models

MalDy framework utilizes a supervised machine learning technique to build its malware investigation models. In this respect, MalDy is composed of a set of models, and each model has a specific purpose. First, we have the threat detection model that finds out the maliciousness likelihood of a given app from its behavioral report. Afterward, the remaining machine learning models aim to investigate individual family threats separately. MalDy uses a model for each possible threat. In our case, we have a malware detection model along with a set of malware family attribution models. In this phase, we build each model separately using X_{build}. All the models are

employing a binary classification to quantify the likelihood of a specific threat. In the process of building MalDy models, we evaluate different classification algorithms to compare their performance. Furthermore, we tune each ML algorithm classification performance under an array of hyper-parameters (different for each ML algorithm). The tunning is a completely automatic process; the investigator only needs to provide X_{build}. We train each investigation model on X_{train} and tune the models performance on X_{valid} by finding the best hyper-parameters as presented in Algorithm 8. Afterward, we determine the optimum decision thresholds for each model using its performance on X_{valid}. At the end of this stage, we have a list of optimum models' tuples $Opt = \{< c_0, th_0, params_0 >, < c_1, th_1, params_1 >, .., < c_c, th_c, params_c >\}$, where c is the number of explored classification algorithms. A tuple $< c_i, th_i, params_i >$ defines the optimum hyper-parameters $params_i$ and decision threshold th_i for ML classification algorithm c_i.

Algorithm 8: Build Models Algorithm

```
Input  : X_build: build set
Output: Opt: optimum models' tuples
X_train, X_valid = X_build
for c in MLAlgorithms do
    score = 0 for params in c.params_array do
        model = train(alg, X_train, params) ;
        s, th = validate(model, X_valid) ;
        if s > score then
            ct = < c, th, params > ;
        end
    end
    Opt.add(ct)
end
return Opt
```

Ensemble Composition

Previously, we discuss the process of building and tunning individual classification models for specific investigation tasks (malware detection, family one threat attribution, family two threat attribution, etc.). In this phase, we construct an ensemble model from a set of models generated using the optimum parameters computed previously (Section 4.3.2), such that the ensemble outperforms any underlying model. We take each set of optimally trained models $\{(C_1, th_1), (C_2, th_2), .., (C_h, th_h)\}$ for a specific threat investigation task and unify them into an ensemble E. The latter utilizes the

weighted majority voting mechanism across the individual model's outcomes for a specific investigation task. Equation 9 shows the computation of the final outcome for one ensemble E, where w_i is the weight given for a single model. The current implementation employs equal weights for the ensemble's models. This phase produces MalDy ensembles, $\{E^1_{Detection}, E^2_{Family1}, E^3_{Family2}$ $.., E^T_{familyJ}\}$, a malware detection ensemble and an ensemble for each malware family.

$$\begin{aligned} \hat{y} = E(x) &= sign\left(\sum_{i}^{|E|} w_i C_i(x, th_i)\right) \\ &= \begin{cases} +1 : \sum_i (w_i C_i) \geq 0 \\ -1 : \sum_i (w_i C_i) < 0 \end{cases} \end{aligned} \tag{9}$$

Ensemble Prediction Process

MalDy prediction process is divided into two phases, as depicted in Algorithm 9. First, given a behavioral report, we generate the feature vector x using TF-IDF or FH vectorization techniques. Afterward, the detection ensemble $E_{detection}$ checks the maliciousness likelihood of the feature vector x. If the maliciousness detection is positive, we proceed to the family threat attribution. Since the family threat ensembles, $\{E^2_{Family1}, E^3_{Family2} .., E^T_{familyJ}\}$, are independent, we compute the outcomes of each family ensemble E_{family_i}. MalDy flags a malware family threat if and only if the majority voting is above a given voting threshold vth (computed using X_{valid}). In the case where no family threat is flagged by the family ensembles, MalDy tags the current sample as an unknown threat. Also, in the case of multiple flagged families, MalDy selects the family with the highest probability, and provides the security investigator with sorted flagged families according to the corresponding likelihood probabilities. The separation between the family attribution models makes MalDy more flexible to update. Adding a new family threat will need only to train, tune, and calibrate the family model without affecting the rest of the framework ensembles.

4.3.3 Framework

In this section, we present the essential techniques used in MalDy framework, namely, N-grams [61], feature hashing (FH), and term frequency-inverse document frequency (TFIDF). Furthermore, we present the explored and tuned machine learning algorithms during the model building phase

Algorithm 9: Prediction Algorithm

Input : $report$: Report
Output: D: Decision
$E_{detection}$ = $E^1_{Detection}$;
E_{family} = $\{E^2_{Family1}, .., E^T_{familyJ}\}$;
x = Vectorize($report$) ;
detection_result = $E_{detection}(x)$;
if *detection_result* < 0 **then**
    return detection_result ;
end
for E_{F_i} *in* E_{family} **do**
    family_result = E_{F_i}(x) ;
end
return detection_result, family_result ;

(Section 4.3.3).

we describe the components of the MalDy related to the automatic security feature engineering process. We compute word N-grams on behavioral reports by counting the word sequences of size N. Notice that N-grams are extracted using a moving forward window (of size N) by one step and incrementing the counter of the found feature (word sequence in the window) by one. The window size N is a hyper-parameter in the MalDy framework. N-gram computation happens simultaneously with the vectorization using FH or TFIDF in the form of a pipeline to prevent computation and memory issues due to the high dimensionality of N-grams. From a security perspective, the N-grams tool can produce distinguishable features between the different variations of an event log compared to single word (1-grams) features. The performance of the malware investigation is profoundly affected by the features generated using N-grams (where $N > 0$). Based on the BoW model, MalDy considers the count of unique N-grams as features that will be leveraged through a pipeline to FH or TFIDF as presented in the previous sections.

Machine Learning Algorithms

Table 4.5 shows the candidate machine learning classification algorithms for MalDy framework. The candidates represent the most used classification algorithms and come from different learning categories, such as decision tree-based learning algorithms. Also, all these algorithms have efficient public implementations. We chose to exclude logistic regression from the list due to the superiority

of SVM in most cases. KNN may consume a lot of memory resources during production because it needs all the training dataset to be deployed in the production environment. However, we keep KNN in the MalDy candidate list because of its unique fast update feature. Updating KNN in a production environment requires only to update the train dataset, and we do not need to retrain the model. This option could be beneficial in certain malware analysis cases.

Classifier Category	Classifier Algorithm	Chosen
Tree	CART	✓
	Random Forest	✓
	Extremely Randomized Trees	✓
General	K-Nearest Neighbor (KNN)	✓
	Support Vector Machine (SVM)	✓
	Logistic Regression	✗
	XGBoost	✓

Table 4.5: Explored Machine Learning Classifiers

4.3.4 Evaluation Results

Evaluation Datasets

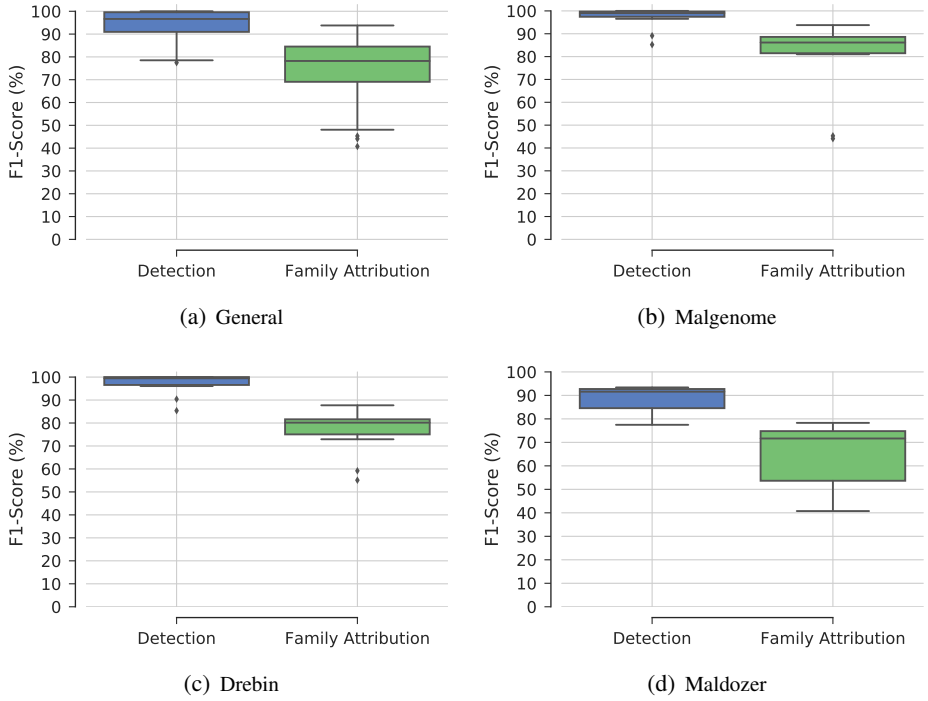
Table 4.6 presents different datasets used to evaluate MalDy framework. We focus on Android and Win32 platforms to prove the portability of MalDy. All the used datasets are publicly available except Win32 Malware dataset, which is provided by a third-party security vendor. Behavioral reports are generated using DroidBox [47] and ThreatAnalayzer[2] for Android and Win32 respectively.

Platform	Dataset	Sandbox	Tag	#Sample/#Family
Android	MalGenome [203]	D	Malware	1k/10
	Drebin [73]	D	Malware	5k/10
	Maldozer [132]	D	Malware	20k/20
	AndroZoo [67]	D	Benign	15k/-
	PlayDrone [3]	D	Benign	15k/-
Win32	Malware [4]	T	Malware	20k/15

Table 4.6: Evaluation Datasets(D: DroidBox, T: ThreatAnalyzer)

Next, we evaluate MalDy on different datasets and various settings. Specifically, we assess the effectiveness of the word-based approach for malware detection and family attribution on Android malware behavior reports. We evaluate the portability and the MalDy concept on other platforms (e.g., Win32 malware) behavioral reports. Finally, we measure the efficiency of MalDy under different machine learning classifiers and vectorization techniques. During the evaluation, we answer

[2]threattrack.com



(a) General (b) Malgenome

(c) Drebin (d) Maldozer

Figure 4.9: MalDy Effectiveness Performance

some other questions related to the comparison between the vectorization techniques, and the used classifiers in terms of effectiveness and efficiency. Also, we show the effect of the training sets size and the usage of machine learning ensemble technique on the framework performance.

Effectiveness

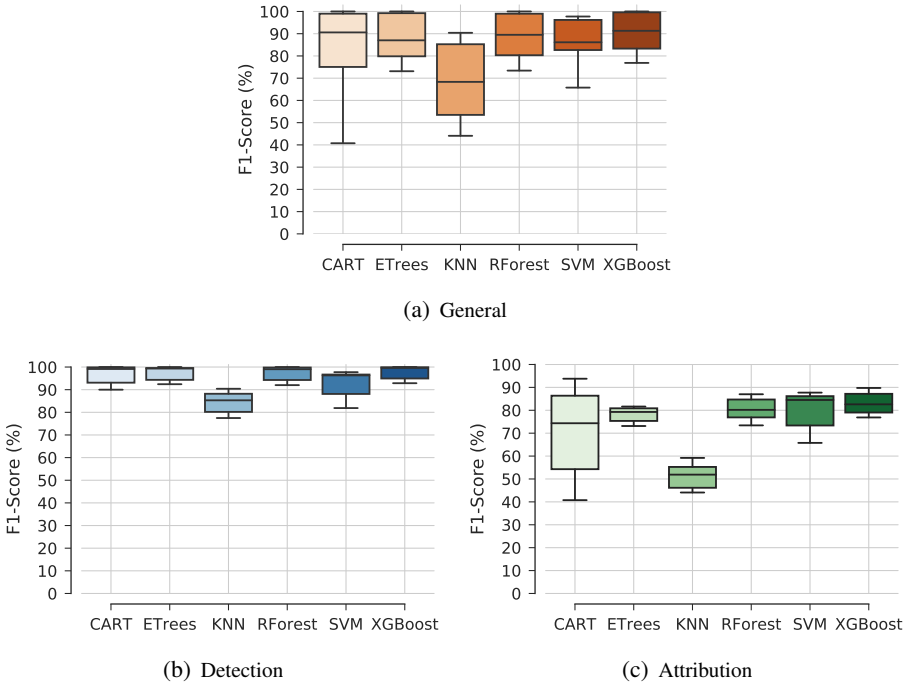
Figure 4.9 shows the detection and the attribution performance under various settings and datasets. The settings refer to the used classifiers in ML ensembles and their hyper-parameters, as shown in Table 4.8. Figure 4.9(a) depicts the overall performance of MalDy. In the detection, MalDy achieves 90% F1-score (100% maximum and about 80% minimum) on average under the various settings (classification models, vectorization techniques, hyper-parameters tuning, single model, and models ensemble). On the other hand, in the family attribution task, MalDy shows over 80% F1-score (family attribution is a harder task than the detection task) in various settings. More granular results for each dataset are shown in Figures 4.9(b), 4.9(c), and 4.9(d) for Malgenome [203], Drebin [73], and Maldozer [132] datasets respectively. Notice that Figure 4.9(a) combines

the performance of baseline (worst performance), tuned, and ensemble models, and summarize the results in Table 4.7.

	Detection (F1 %)			Attribution (F1 %)		
	Base	**Tuned**	**Ens**	**Base**	**Tuned**	**Ens**
General						
mean	86.06	90.47	94.21	63.42	67.91	73.82
std	6.67	6.71	6.53	15.94	15.92	14.68
min	69.56	73.63	77.48	30.14	34.76	40.75
25%	83.58	88.14	90.97	50.90	55.58	69.07
50%	85.29	89.62	96.63	68.81	73.31	78.21
75%	91.94	96.50	99.58	73.60	78.07	84.52
max	92.81	97.63	100.0	86.09	90.41	93.78
Genome						
mean	88.78	93.23	97.06	71.19	75.67	79.92
std	5.26	5.46	4.80	16.66	16.76	16.81
min	77.46	81.69	85.23	36.10	40.10	44.09
25%	85.21	89.48	97.43	72.36	77.03	81.47
50%	91.82	96.29	99.04	76.66	81.46	86.16
75%	92.13	96.68	99.71	80.72	84.82	88.61
max	92.81	97.63	100.0	86.09	90.41	93.78
Drebin						
mean	88.92	93.34	97.18	65.97	70.37	76.47
std	4.93	4.83	4.65	9.23	9.14	9.82
min	78.36	83.35	85.37	47.75	52.40	55.10
25%	84.95	89.34	96.56	61.67	65.88	75.05
50%	91.60	95.86	99.47	69.62	74.30	80.16
75%	92.25	96.53	100.0	72.68	76.91	81.61
max	92.78	97.55	100.0	76.28	80.54	87.71
Maldozer						
mean	80.48	84.85	88.38	53.11	57.68	65.06
std	6.22	6.20	5.95	16.03	15.99	13.22
min	69.56	73.63	77.48	30.14	34.76	40.75
25%	75.69	80.13	84.56	39.27	43.43	53.65
50%	84.20	88.68	91.58	56.62	61.03	71.65
75%	84.88	89.01	92.72	67.34	71.89	74.78
max	85.68	89.97	93.39	71.17	76.04	78.30

Table 4.7: Tuning Effect on Performance

Classifier Effect. The results in Figure 4.10, Table 4.7, and Table 4.8 confirm the effectiveness of MalDy framework and its word-based approach. Figure 4.10 presents the effectiveness performance of MalDy using the different classifiers for the final ensemble models. Figure 4.10(a) shows the combined performance of the detection and family attribution. All the ensembles achieved a good F1-score, and XGBoost ensemble shows the highest scores. Figure 4.10(b) confirms the previous scores for the detection task. Also, Figure 4.10(c) presents the malware family attribution scores per ML classifier. More details on classifiers performance is depicted in Table 4.8.



(a) General

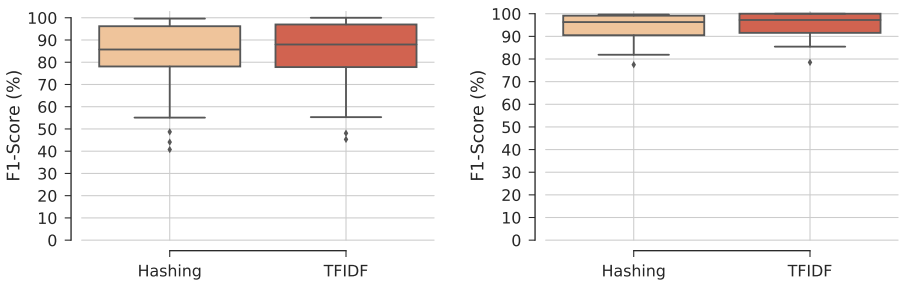
(b) Detection

(c) Attribution

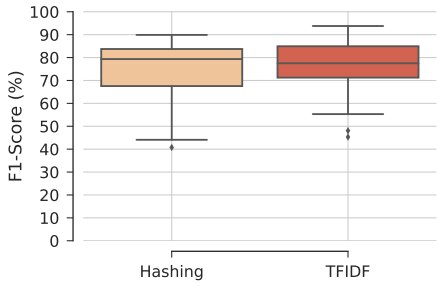
Figure 4.10: Effectiveness per Machine Learning Classifier

Effect of the Vectorization Technique. Figure 4.11 shows the effect of vectorization techniques on the detection and the attribution performance. Figure 4.11(a) depict the overall combined performance under various settings. As depicted in Figure 4.11(a), Feature hashing and TF-IDF show a very similar performance. In the detection task, the vectorization techniques' F1-scores are almost identical to those presented in Figure 4.11(b). We notice a higher overall attribution score using TF-IDF compared to FH, as shown in Figure 4.11(c). However, there are some cases where FH outperforms TF-IDF. For instance, XGBoost achieves a higher attribution score under feature hashing vectorization, as shown in Table 4.8.

Effect of Tuning Hyperparameters. Figure 4.12 illustrates the effects of tuning and ensemble phases on the overall performance of MalDy. In the detection task, as in Figure 4.12(a), the ensemble improves the performance by 10% (F1-score) over the base model. The ensemble is composed of a set of tuned models that already outperform the base model. In the attribution task, the ensemble improves the F1-score by 9%, as shown in Figure 4.12(b).

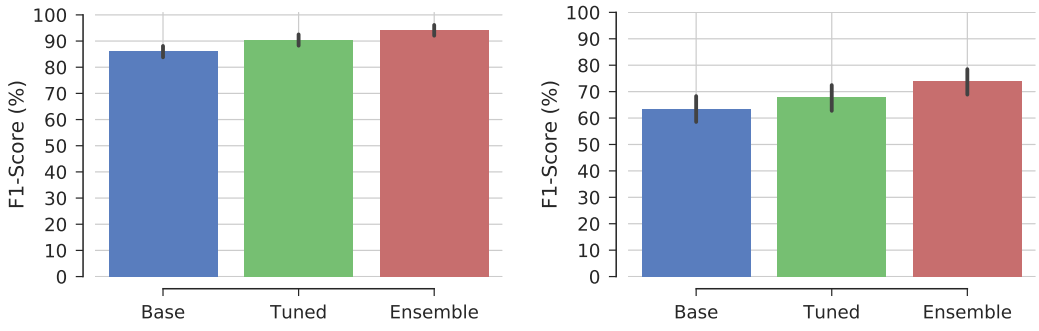


(a) General (b) Detection



(c) Attribution

Figure 4.11: Effectiveness per Vectorization Technique



(a) Detection (b) Attribution

Figure 4.12: Ensemble Performance and Tunning Effect

Portability

In the following, we assess the portability of the MalDy by applying the framework on a new type of behavioral reports. Also, we investigate the appropriate training dataset size for MalDy to achieve a good results. We report only the results of the attribution task on Win32 malware because currently we do not have a dataset of Win32 benign behavioral reports for the detection task.

Settings			**Attribution** F1-Score (%)			**Detection** F1-Score (%)			
Model	**Dataset**	**Vector**	**Base**	**Tuned**	**Ensemble**	**Base**	**Tuned**	**Ensemble**	**FPR**(%)
cart	drebin	hashing	64.93	68.94	72.92	91.55	95.70	99.40	00.64
	drebin	tfidf	68.12	72.48	75.76	92.48	96.97	100.0	00.00
	genome	hashing	82.59	87.28	89.90	91.79	96.70	98.88	00.68
	genome	tfidf	86.09	90.41	93.78	92.25	96.50	100.0	00.00
	maldozer	hashing	33.65	38.56	40.75	82.59	87.18	90.00	06.92
	maldozer	tfidf	40.14	44.21	48.07	83.92	88.67	91.16	04.91
etrees	drebin	hashing	72.84	77.27	80.41	91.65	95.77	99.54	00.23
	drebin	tfidf	71.12	76.12	78.13	92.78	97.55	100.0	00.00
	genome	hashing	74.41	79.20	81.63	91.91	96.68	99.14	00.16
	genome	tfidf	73.83	78.65	81.02	92.09	96.61	99.57	00.03
	maldozer	hashing	65.23	69.34	73.13	84.56	88.70	92.42	06.53
	maldozer	tfidf	67.14	71.85	74.42	84.84	88.94	92.74	06.41
knn	drebin	hashing	47.75	52.40	55.10	78.36	83.35	85.37	12.86
	drebin	tfidf	51.87	56.53	59.20	82.48	86.57	90.40	05.83
	genome	hashing	36.10	40.10	44.09	77.46	81.69	85.23	07.01
	genome	tfidf	37.66	42.01	45.31	81.22	85.30	89.13	02.10
	maldozer	hashing	41.68	46.67	48.69	69.56	73.63	77.48	26.21
	maldozer	tfidf	48.02	52.73	55.31	70.94	75.36	78.51	03.86
rforest	drebin	hashing	72.63	76.80	80.46	91.54	95.95	99.12	00.99
	drebin	tfidf	72.15	76.40	79.91	92.31	96.62	100.0	00.00
	genome	hashing	78.92	83.73	86.12	91.37	95.79	98.95	00.68
	genome	tfidf	79.45	83.90	87.00	92.75	97.49	100.0	00.00
	maldozer	hashing	66.06	70.72	73.41	84.49	88.96	92.01	07.37
	maldozer	tfidf	67.96	72.04	75.89	85.07	89.41	92.72	06.10
svm	drebin	hashing	57.35	61.95	82.92	84.50	89.33	96.08	00.86
	drebin	tfidf	63.11	67.19	87.71	85.11	89.35	96.73	01.15
	genome	hashing	69.99	74.68	86.08	85.47	89.83	96.54	00.19
	genome	tfidf	73.16	77.82	86.20	84.46	88.46	97.73	00.39
	maldozer	hashing	30.14	34.76	65.76	72.32	77.12	81.88	15.82
	maldozer	tfidf	36.69	41.09	70.18	76.82	81.14	85.46	08.56
xgboost	drebin	hashing	76.28	80.54	**84.01**	92.05	96.50	**99.61**	**00.29**
	drebin	tfidf	73.53	77.88	**81.18**	92.23	96.45	**100.0**	**00.00**
	genome	hashing	81.80	85.84	**89.75**	91.86	96.09	**99.62**	**00.32**
	genome	tfidf	80.36	84.48	**88.24**	92.81	97.63	**100.0**	**00.00**
	maldozer	hashing	71.17	76.04	**78.30**	85.68	89.97	**93.39**	**05.86**
	maldozer	tfidf	69.51	74.15	**76.87**	85.01	89.16	**92.86**	**06.05**

Table 4.8: Android Malware Detection

MalDy on Win32 Malware. Table 4.9 presents MalDy attribution performance in terms of F1-score. In contrast with previous results, we trains MalDy models on only 2k (10%) out of 20k report' dataset (Table 4.6). The rest of the reports are used for testing (18k reports, or 80%). Despite that, MalDy achieves high scores that reach 95%. The results in Table 4.9 illustrate the portability of MalDy, which increases the utility of the framework across the different platforms and environments.

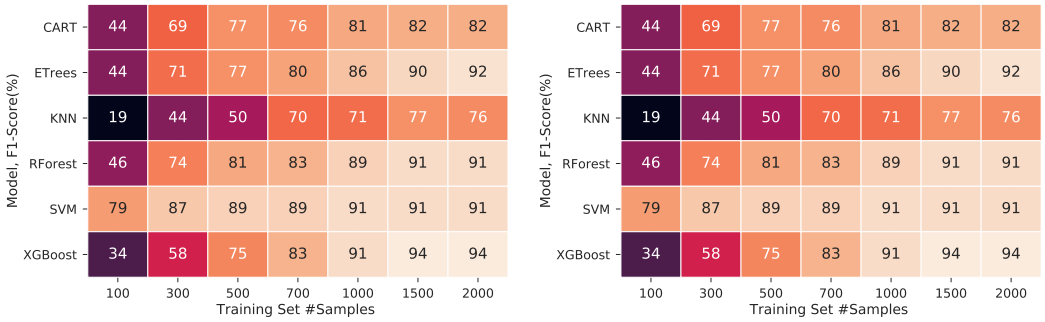
Model	Ensemble F1-Score(%)	
	Hashing	TFIDF
CART	82.35	82.74
ETrees	92.62	92.67
KNN	76.48	80.90
RForest	91.90	92.74
SVM	91.97	91.26
XGBoost	**94.86**	**95.43**

Table 4.9: System Performance on Win32 Malware

MalDy Train Dataset Size Using the Win32 malware dataset (Table 4.9), we investigate the training set size hyper-parameter for MalDy to achieve good results. Figure 4.13 illustrates the outcome of our analysis for both vectorization techniques and the different classifiers. We notice the high scores of MalDy even with relatively small datasets. This is made very clear in the case where MalDy uses the SVM ensemble, which corresponds to a 87% F1-score with only 200 training samples.

Efficiency

Figure 4.14 illustrates the efficiency of MalDy by showing the average runtime require to investigate a behavioral report. The runtime is composed of the preprocessing time and the prediction time. As depicted in Figure 4.14, MalDy needs only about 0.03 seconds per given report for all the ensembles and the preprocessing settings except for the SVM ensemble. The latter requires 0.2 to 0.5 seconds (depending on the preprocessing technique) to decide about a given report. Although the SVM ensemble needs a small training set to achieve good results, it is costly in production environment in terms of runtime. Therefore, the security investigator could customize the MalDy to suit particular analysis priorities. The efficiency experiments have been conducted on Intel(R) Xeon(R) CPU E52630 (128G RAM), using only one CPU core.



(a) Hashing (F1-Score %)

(b) TF-IDF (F1-Score %)

Figure 4.13: Win32 Performance and Effect of the Training Size

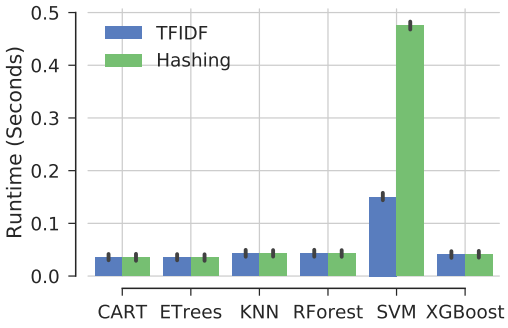


Figure 4.14: Overall Framework Efficiency

4.4 Summary

In this chapter, we detailed a pioneering investigation on the use of dynamic features for Android malware fingerprinting. We leverage state-of-the-art *machine learning* and NLP techniques to produce generic fingerprints. The evaluation of DySign on both real-life malware and benign apps demonstrated good detection and attribution performances with high scalability. By using the DySign concept, we proposed a portable, effective, and yet efficient framework for malware detection and family attribution. The key concept involves the modeling of behavioral reports using the bag of words model. Furthermore, we leverage advanced NLP and ML techniques to build discriminative machine learning ensembles. MalDy achieves over 94% F1-score in Android malware detection task on Malgenome, Drebin, and MalDozer datasets and more than 90% in the malware

family attribution task. We demonstrate MalDy portability by applying the framework on Win32 malware reports where the framework achieves 94% on the attribution task. MalDy performance depends on the execution environment reporting system, and the quality of the reporting affects its performance.

In the previous chapter and the current one, we focus on Android malware detection using machine learning classification and clustering, respectively based on dynamic and static analyses features. In the next chapter, we propose a system for the mining of cyber-threat networks, which are composed of malicious IP addresses and domain names, starting from the detected Android malware. The aforementioned cyber-threat networks support malicious apps and act as a backend service. Thus, finding malicious cyber-threat networks represents a natural progression following the detection of Android malware as proposed in previous chapters (Chapters 4 and 3).

Chapter 5

Fingerprinting Cyber-Infrastructures of Android Malware

5.1 Overview

In this chapter, we propose ToGather, an automatic investigation framework for Android malware cyber-infrastructures. In our context, a malware cyber-infrastructure is a set of IP addresses and domain names orchestrated together to serve as a backend for malicious activities, including malicious apps. ToGather framework is a set of techniques and tools together with security feeds, which automatically build a situational awareness about Android malware cyber-infrastructures. ToGather characterizes the cyber-infrastructure starting from android malware samples to relate the malware to the corresponding network footprint in terms of IPs and domains. ToGather goes even a step further by dividing this cyber-infrastructure into sub-infrastructure components based on the connectivity between nodes. The result is in the segmentation of the global threat network into multiple network communities representing many granular sub-cyber-infrastructures. To this end, ToGather leverages cyber-threat intelligence that is derived from various sources such as spam, Windows malware, darknet, and passive DNS to ascribe cyber-threats to the corresponding cyber-infrastructure. Accordingly, the input of ToGather framework is made of malware samples togather with security feeds, while the output represents networks of cyber-infrastructures together with their

network footprint, which provides the security practitioner an overview of Android malware cyber-activities on the Internet.

5.1.1 Threat Model

We position ToGather as a detector of malicious cyber-infrastructures of Android malware. It is designed to uncover threat networks and sub-networks starting from Android malware samples. ToGather does not guarantee zero false-positives due to the large number of benign domain names and IP addresses that might not be filtered out using ToGather whitelists. ToGather is resilient to obfuscation during the extraction of network information from Android malware because it applies both static and dynamic analyses. Hence, if the static content is heavily obfuscated, ToGather is still able to collect IP addresses and domain names from dynamic analysis reports.

5.1.2 Usage Scenarios

ToGather is designed to be practical and efficient in the hands of security practitioners. (1) Security analysts might use ToGather framework as an investigation tool to minimize the efforts of generating threat networks for a given Android malware family. The analyst leverages the IP addresses and domain names ordered by their importance in the generated threat network to prioritize the takedown and mitigation operations. (2) ToGather acts as a monitoring system. It analyzes a feed of Android malware (e.g., new samples daily) to generate a snapshot of the threat network and thus uncover malicious activities (e.g., spamming and phishing). Periodic reporting gives insights into the evolution and the malicious behaviors of a given malware family over time.

5.2 Methodology

In this section, we present the overall workflow of ToGather framework, as shown in Figure 5.1, starting from Android malware samples and ending with the produced relevant threat intelligence:

1) The first step in ToGather consists of deriving network information from Android samples in a given analysis window (e.g., day, week, month) whether the samples are from the same malware family or not. However, we consider one malware family as a typical use-case of ToGather, as

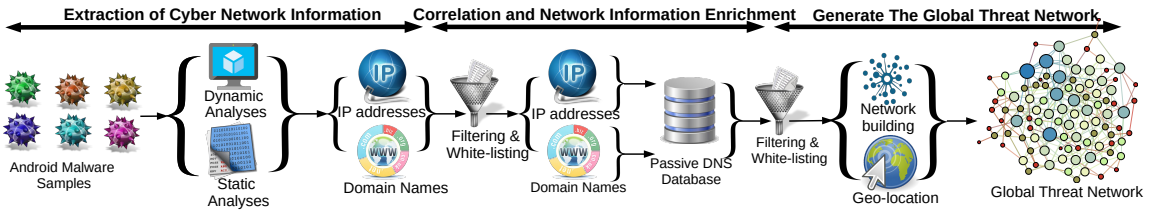


Figure 5.1: ToGather Approach Overview

presented in the evaluation section. ToGather conducts dynamic and static analyses where each analysis produces a report for each Android malware sample. Therefore, we produce dynamic and static analyses reports for each malware sample. Leveraging both analysis types enhances the resiliency of ToGather against common obfuscation techniques, which hide relevant information about malicious activities such as domain names and IP addresses (network information). Afterward, ToGather extracts network information (IP addresses and domain names) by parsing the related text blocks (strings) from analysis reports and applies a simple text pattern search. In static analysis, we mainly concentrate on the Dalvik compiled code (classes.dex) for such extraction. We collect network information more efficiently from dynamic analysis reports since they are more structured and have labeled fields.

2) Next, we filter the extracted network identifiers from noise information such as non-routed IP addresses. Also, we filter domain names and URLs that use Unicode characters. For the current ToGather implementation, we consider domain names and URLs written only in the standard English/Latin alphabet. In the case of URL links, we keep only domains. To this end, we have a set of valid IP addresses and domain names found in Android malware. It is important to notice that malware hashs tag network information, and these tags are kept during all the workflow steps of ToGather. To minimize false positives, ToGather applies whitelisting mechanisms. For domain names, ToGather leverages Alexa [51] and Quantcast [53] (more than one million domain names). However, the number of white domain names is a hyper-parameter of ToGather that can be used to control the number of false positives. In the case of IP addresses, we leverage a set of public white IPs such as Google DNS servers and other ones [20]. It is important to emphasize that ToGather considers public cloud vendor IPs and domain names as a whitelist. The aim is to observe and then gain insight into the use of the cloud infrastructure by Android malware. This idea originates from

the observation that Android malicious apps (and malware in general) make more use of the cloud as a low-cost infrastructure for their malicious activity.

3) In this step, we propose a mechanism to enhance and enrich the malicious network information to cover related domains and IPs. In essence, ToGather aims at answering the following questions: (i) What are the IP addresses of current malicious domains? Here we investigate the IP addresses of server machines that host malicious activities that are most likely related to the analyzed Android malware. (ii) What are the domain names pointing to the current malicious IP addresses? The intuition is that a malicious server machine with a given IP address could host various malicious contents, and the adversary could use multiple domains pointing to such contents. To answer this question, ToGather has a module to enrich network information using passive DNS replication. The latter is a technology that builds zone replicas without the cooperation from zone administrators, based on captured name server responses, as presented in Section 5.2.3. We use the network information, whether IP addresses or domains, as parameters of two functions applied on a passive DNS database. The goal of the function is to enrich the list of domains and IP addresses that could be part of the adversary threat network. The enrichment services are: (i) GetIP(Domain): This function takes a domain as a parameter to query the passive DNS database. The result is all IP addresses pointing to the domain. (ii) GetDomain(IP): This function gets all the domains that resolve to the IP address given as a parameter.

We consider passive DNS correlation for two reasons: (i) A small number of Android malware samples generally yields limited network information. (ii) Security practitioners aim at having a more in-depth situational awareness about malware Internet activity. As such, they would like to consider all related IPs and domain names. The result of the correlation is a set of IP addresses and domain names inferred using passive DNS related to Android malware apps. The correlation results could, however, overwhelm the investigation process. Passive DNS correlation is therefore optional if we have a significant number of samples from a given Android family. The correlation with passive DNS could produce some known benign entries. For this reason, we filter the likely harmless network information by matching the newly found IP addresses against top Alexa [51] and Quantcast [53] domain names and known public IP addresses [52].

4) At this stage, we have a set of network information tagged by malware hashes. To extract

relevant and actionable intelligence, ToGather aggregates all the previous records into a heterogeneous network with different types of nodes: *malware hashes*, *IP addresses* and *domain names*. We consider the heterogeneous network that is extracted from a given Android malware family as the malicious activity map of that family on the Internet. We call such a heterogeneous network, a *threat network*. Furthermore, ToGather produces homogenous networks by executing multiple projections according to the node type (IP address or domain name). Therefore, ToGather produces three homogeneous graphs, one only considers IP addresses connections, the other only considers domain name connections, and a threat network with IPs and domains as network information. The Graph homogeneity is required to apply graph partitioning on domain threat network, and network information threat network.

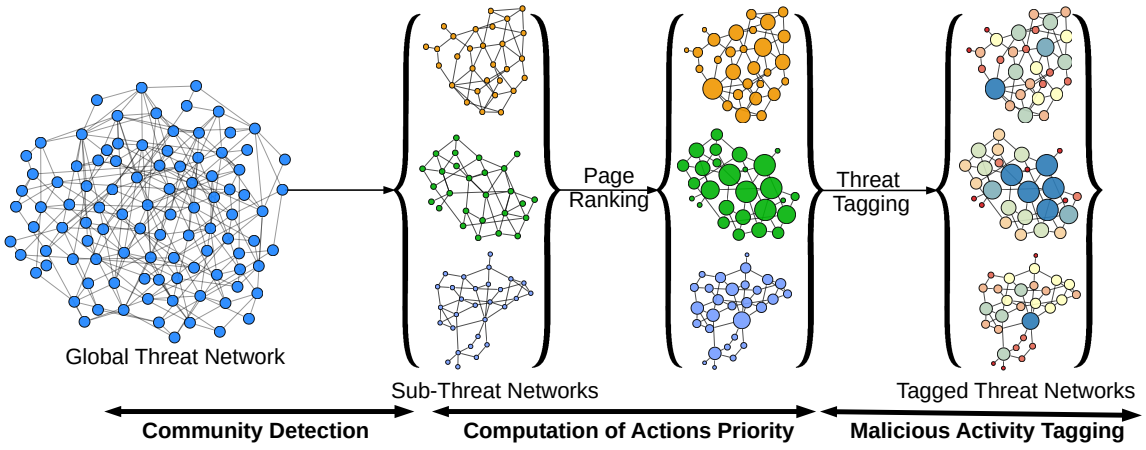


Figure 5.2: Graph Analysis Overview

5) Further, ToGather aims at producing more granular graphs (see Figure 5.2) from the generated threat networks derived in the previous step. In this respect, ToGather checks the possibility of community identification in these threat networks based on the connectivity between nodes. The higher is the connectivity between the nodes in a particular area of the network, the more is the possibility to have a malicious community. For community detection (Section 5.2.1), we adopt a highly scalable algorithm [82] to enhance ToGather community detection module. The intuition behind using the community concept is as follows: (i) Considering ToGather typical usage scenario, where we enter Android malicious apps from the same family, the community could define different threat networks that are related to the malicious activities. In other words, either one adversary is using

these threat networks as backups, or we have multiple adversaries instead. In the case of Android malware, the second hypothesis is more plausible because of the low cost of repackaging of existing malware samples to suit the need of the adversary. (ii) In case ToGather receives Android malware from different families, the communities is interpreted as the threat networks of different Android malware families to focus on the relation between them. The output of this step is a set of threat networks related to IPs, domains, as well as network information and their communities (sub-threat networks).

6) To produce actionable cyber-threat intelligence, we leverage the page ranking algorithm (Section 5.2.2) to deliver ranking scores for critical nodes of a given (sub)-threat network. Consequently, the investigator should have some priority list when it comes to mitigation or takedown of nodes that are associated with a malicious cyber-infrastructure. As a result, ToGather produces a threat network for each Android malware family together with the ranking of each node. Because ToGather generates multiple homogeneous graphs based on the node type (IP, domain, network information), it produces different ranking lists. Therefore, the security practitioner has the opportunity of selecting the node type during the mitigation or the takedown to protect his system. Also, it is essential to mention that it is expensive for the adversary to get new IP addresses. In contrast, domain names could be frequently changed due to their affordability.

7) We do not focus only on Android malware. Instead, we aim to gain insights into the shared network IP and domains of the analysed Android malware samples with other platform malware families. Indeed, an adversary could have many malicious activities in several operating systems to achieve wider coverage. Therefore, similarly to the first step, we conduct dynamic and static analyses on Windows and Linux malware samples to extract the corresponding network information. The same step is applied to this network information. Afterward, we correlate the Android network information with the non-Android malware information to discover another dimension of the adversary network. The result will be all IP addresses and domains of Android malware in addition to all network records of a given non-Android malware family if they share some network information. It is essential to notice that malware families also label information networks of non-Android malware.

8) In this final workflow step of ToGather, we leverage other intelligence sources to label malicious activities that are committed by the discovered threat networks. The current ToGather implementation includes the correlation with spam emails, reconnaissance traces, and phishing URLs. We consider ToGather as an active service that receives at every epoch time (day, week, month) Android malware with the corresponding family (the typical use case) and produces valuable intelligence about this malware family.

5.2.1 Threat Communities Detection

A scalable community detection algorithm is essential to extract communities from the threat network. For this reason, we empower ToGather with the Fast Unfolding Community Detection algorithm [82], which can scale to billions of network links. The algorithm achieves excellent results by measuring the *modularity* of communities. The latter is a scalar value $M \in [-1, 1]$ that measures the density of edges inside a given community compared to the edges between communities. The algorithm uses an approximation of the modularity since finding the exact value is computationally hard [82]. The main reason to choose the algorithm proposed in [82] is its scalability. As depicted in Figure 5.3, we apply the community detection on a million-node graph with a medium density ($P = 0.001$ probability of connecting a node A is another node B in the generated network), which we believe has a similar density to the threat network generated from Android malware samples. For the sake of completeness, we perform the same experiment on graphs with a different probability P. As presented in Figure 5.4(c), we can detect communities in $30,000$-node graphs with ultra high density (unrealistic) in a relatively small (compared to the time dedicated to the investigation) amount of time (3 hours).

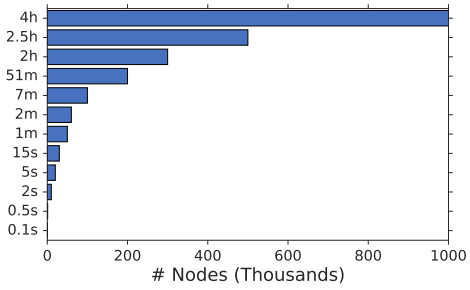
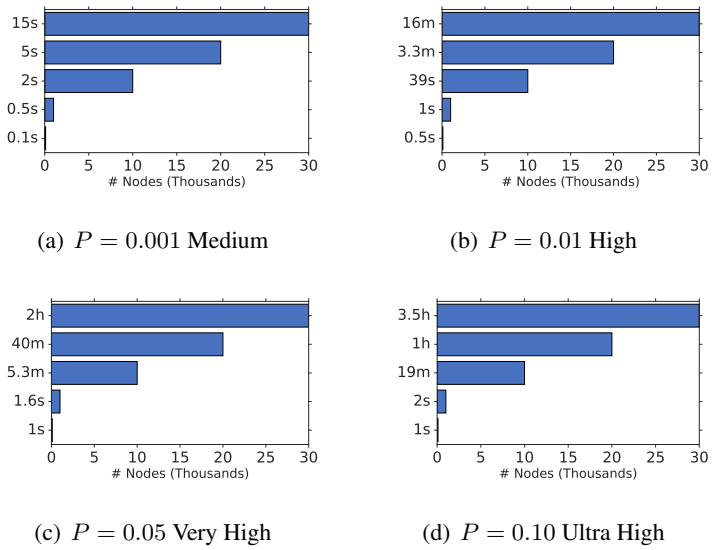


Figure 5.3: Scalability of the Community Detection



(a) $P = 0.001$ Medium

(b) $P = 0.01$ High

(c) $P = 0.05$ Very High

(d) $P = 0.10$ Ultra High

Figure 5.4: Graph Density Versus Scalability

The previous algorithm requires as input a homogeneous network to work correctly. In our case, the threat network generated from the network information is heterogeneous because it contains two main node types: (i) The malware sample identifier, which is the cryptographic hash of the malware sample; (ii) the network information: the domain names and the IPv4 addresses. Also, we apply the projection on the first heterogeneous network to generate homogeneous graphs. To do so, ToGather makes the graph projection by abstracting away from malware identifier while keeping network information, i.e., if malware connects to two IPs, the projection would produce only the two IPs involved in the connection. To this end, we get different projection results based on the node abstraction: (a) General threat network containing both IP addresses and domain names, (b) IP threat network containing only IP addresses, (c) domains related threat network containing only domain names.

5.2.2 Action Prioritization

From community detection, ToGather checks if there are possible sub-graphs in the threat networks based on node connectivity. Even though threat networks zoom into malicious cyber-infrastructures of a given Android malware family, it is difficult for the security practitioner to tackle

the whole threat network at once. For this reason, ToGather proposes an action priority system. The latter takes the IP, domain (or both), and the threat network and produces an action priority list based on the maliciousness of each node. By leveraging the graph structure of the threat network, we measure the maliciousness of a given node by its degree, meaning the number of edges that connect it to other nodes. From a security point of view, the more connections an IP or domain has, the more it is important for a malicious cyber-infrastructure. Therefore, we build a priority list sorted by the importance, that an IP, or a domain can inflict in terms of malicious activity. The importance of a given node in a network graph is known as *node's centrality*. This represents a real-valued function tailored to provide a ranking [83] which identifies the most relevant nodes. For this purpose, some algorithms have been defined, such as Hypertext Induced Topic Search (HITS) algorithm [140] and Google's PageRank algorithm [84]. In our approach, we adopt Google's PageRank algorithm due to its efficiency [154]. In the following, we briefly introduce the PageRank algorithm and the random surfer model.

PageRank Algorithm

Definition 1 *(PageRank).*

Let $I(v_i)$ be the set of vertices that link to a vertex v_i and let $deg_{out}(v_i)$ be the out-degree centrality of a vertex v_i. The PageRank of a vertex v_i, denoted by $PR(v_i)$, is provided by the following [84]:

$$PR(v_i) = d \left[\sum_{v_j \in I(v_i)} \frac{PR(v_j)}{deg_{out}(v_i)} \right] + (1-d)\frac{1}{|D|} \tag{10}$$

The constant d is called *damping factor*, assumed to be set to 0.85 [84]. The previous Equation breaks down to one equation per node v_i with an equal number of unknown $PR(v_i)$ values. The PageRank algorithm tries to find out iteratively different PageRank values, which sum up to 1 ($sum_{i=1}^{n} PR(v_i) = 1$). The authors of the PageRank algorithm consider the use case of web surfing, where the user starts from a web page and randomly moves to another one through a web link. If the web surfer is on page v_j with a probability or a damping factor d, then the probability to change page v_i is $\frac{1}{deg_{out}(v_j)}$. The user could follow the links and teleport [84] to a random web page in V

with $1 - d$ probability. The described surfing model is a stochastic process, and W is a stochastic transition matrix, where node ranking values are computed as presented in the following:

$$\vec{PR} = d\left[W.\vec{PR}\right] + (1-d)\frac{1}{|D|}\vec{1} \tag{11}$$

The stochastic matrix W is defined as follows:

$w_{ij} = \frac{1}{deg_{out}(v_j)}$ if a vertex v_j is linked to v_i
$w_{ij} = 0$ otherwise

The notation $\vec{R}$ stands for a vector where its i_{th} element is $PR(v_i)$ (PageRank of v_i). The notation $\vec{1}$ stands for a vector having all elements equal to 1. The computation of PageRank values is done iteratively by defining a convergence stopping criterion ϵ. At each computation step t, a new vector $(\vec{PR}, t)$ is generated based on previous vector values $(\vec{PR}, t-1)$. The algorithm stops computing values when the condition $|(\vec{PR}, t) - (\vec{PR}, t-1)| < \epsilon$ is satisfied.

5.2.3 Security Correlation

Network Enrichment Using Passive DNS

A DNS sensor [187] is used to capture inter-server DNS communication in a passive DNS database. Afterward, the records of passive DNS stored in the database can be queried. We can benefit from a passive DNS database in many ways. For instance, we can know the history of a domain name, as well as the IP addresses that the domain is or was pointing to. We can also find out what domain names are hosted on a given name server or what domains are/(have been) pointing to a given IP address. There are a lot of use cases of passive DNS for security purposes (e.g., mapping criminal cyber-infrastructure [72], tracking spam campaigns, tracking malware command and control systems, detection of fast-flux networks, security monitoring of a given cyber-infrastructure and botnet detection). In our context, we propose the correlation of ToGather intelligence with a passive DNS database to enrich the investigation of Android malware by (i) Finding suspicious

domains that are pointing to a malicious IP address extracted from the analysis of a malware sample; (ii) finding suspicious IP addresses that are resolved from a malicious domain that is obtained from the analysis of malware sample; (iii) measuring the maliciousness magnitude of an IP. The maliciousness can be measured by counting the number of domains that resolve to this malicious IP address. Typically, these domains could be related to different malicious activities or a single one; (iv) filtering outdated domain names: The passive DNS query generally returns timestamp information. ToGather could leverage timestamps to filter out old domain names that are no longer active.

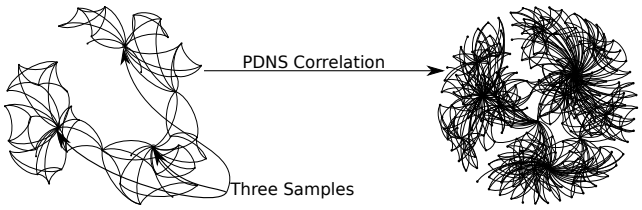


Figure 5.5: Threat Network With/Without Correlation

Threat Network Tagging

From Android malware samples, ToGather produces a threat network that summarizes their malicious activities. Afterward, ToGather detects and provides threat sub-networks if any. Besides, it helps prioritizing the actions to be taken to mitigate malicious activities using the PageRank algorithm. In this section, we go a step further towards the automatic investigation by leveraging other security feeds. Specifically, we aim at correlating threat networks with spam intelligence, reconnaissance intelligence, etc. The objective is to give a multi-dimensional view of the malicious activities that are related to the investigated Android malware family. Moreover, ToGather considers the correlation with network information from other platform malware; in the current setup, we correlate with PC malware from different operating systems.

PC Malware: ToGather tags every produced threat network by leveraging a database of network information extracted from PC malware VirusShare [5]. The malware database is continuously updated. The obtained data is identified by malware hash and its malware family. The latter helps

identifying PC malware (and their families) that share network information with the Android threat network.

Spam: ToGather takes advantage of a spam database (30 Million records) to report the relationship between spamming campaigns and a given threat network. This information is precious for security analysts who are tracking spam campaigns.

Phishing: Similarly to spamming, we consider phishing activities in ToGather tagging. Phishing activities aim at stealing sensitive information using fake web pages that are similar to known trusted ones. Typically, the attacker spreads phishing sites using malicious URLs. We extract only the domain name and store it in a phishing database (5 Million records).

Probing: ToGather considers tags of the threat network nodes if they are part of a probing activity. This presupposes the availability of a probing database (300 Million records) that contains IP addresses that have been part of scanning activities within the same epoch. Probing is derived from darknet traffic, and the probing IP addresses could be persisted in a probing database.

5.3 Experimental Results

In this section, we present the evaluation results of our proposed system. The goal of the evaluation is to assess the effectiveness of ToGather framework in terms of its ability to provide cyber-threat situational awareness from a set of Android malware samples.

5.3.1 Android Malware Dataset

In the evaluation, we use a real Android malware dataset from Drebin [73], a known dataset that contains samples labeled with their families. Drebin dataset [18] contains 5560 labeled malware samples from 179 families [18], as shown in Table 5.1. It is important to stress that Drebin contains all the samples of the MalGenome dataset [17]. As a ground truth for the malware labeling, we take the labels provided by Drebin since there are some differences between Genome and Drebin dataset

labeling. For example, MalGenome recognizes different versions of DroidKungFu malware (1, 2, and 4), where Drebin has only DroidKungFu.

	Malawre Family	Number of Samples
0	FakeInstaller	925
1	DroidKungFu	667
2	Plankton	625
3	Opfake	613
4	GinMaster	339
5	BaseBridge	330
6	Iconosys	152
7	Kmin	147

Table 5.1: Dataset Description by Malware Family

5.3.2 Implementation

We have implemented ToGather using *Python* programming language. In the static analysis, in order to perform reverse engineering of the *Dex* byte-code, we use *dexdump*, a tool provided with Android SDK. We extract the network information from the *Dex* dis-assembly using regular expressions. Beside, ToGather extracts network information from static text content in the APK file of Android malware.

In dynamic analysis, a cornerstone of ToGather is the sandboxing system, which heavily influences the produced analysis reports. We use *DroidBox* [47], a well-established sandboxing environment based on the Android software emulator [12] provided by Google Android SDK [33] as presented in the previous chapter.

5.3.3 Drebin Threat Network

In this section, we present the results of applying ToGather framework on the samples of Drebin dataset with all the 179 families. Figure 5.6 depicts the threat network information (domain names and IP addresses) of Drebin dataset, where a different color represents each family. Although the threat networks are not very clear visually, we could distinguish some connected communities with the same nodes' colors, i.e., the same malware family. This initial observation enhances the need for a community detection module in ToGather. The community here is a set of graph nodes that are highly connected even though they share some links with external nodes. In Figure 5.7, we consider only domain names; here, we can distinguish more sub-threat networks having nodes

from the same malware family. We choose to filter all IP addresses for Drebin dataset due to observations during the experimentation process: (i) Some malware samples contain a significant number of IP addresses; exceeding, in some cases, 100 IPs such as Plankton sample with MD5 hash `3f69836f64f956a5c00aa97bf1e04bf2`. The adversary could deceive the investigator by overwhelming the app with unused IP addresses along with used ones. (ii) A big portion of the IP addresses are part of cloud infrastructure; we filter most of the public ones, but there are plenty of less known infrastructures in remote countries. (iii) In most cases, the adversary utilizes domains for malicious activity due to the low cost and flexibility compared to IP addresses. In this experimentation, we consider only domain names, but the security analyst could include IP addresses when needed.

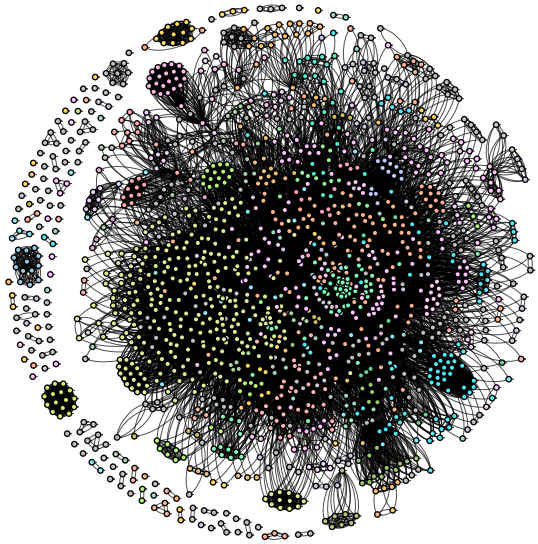


Figure 5.6: Network Information of Drebin Dataset

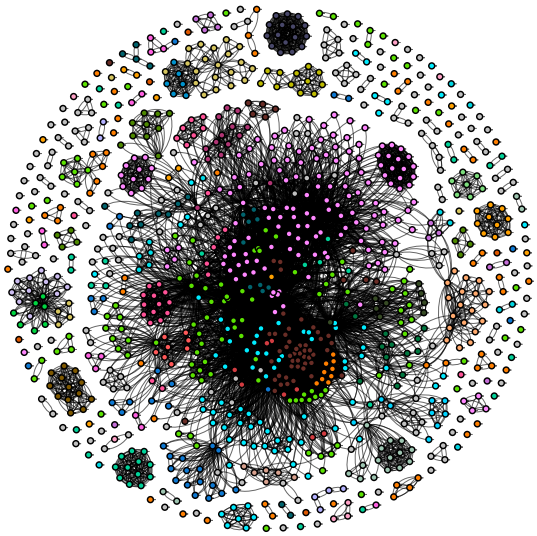
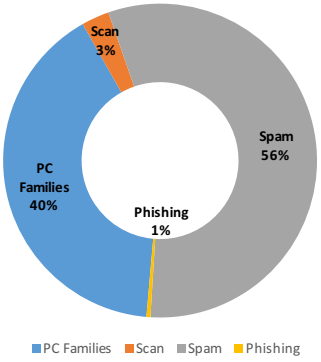


Figure 5.7: Domain Names Drebin Dataset

Using all Drebin dataset (179 malware families) to produce the threat network is an extreme use case for ToGather framework; using only few malware families represents a typical use case when we aim to investigate the threat network relations. However, even with the whole Drebin dataset, ToGather, as presented in Figure 5.7, shows promising results, where we notice sub-threat networks with/without links to other nodes. By considering only domain names in Figure 5.7, it is noticeable that the size of the threat network significantly decreases by removing IP addresses; typically, there are substantially more domains than IP addresses in the Android apps. However, this is due to the extensive whitelisting of domains compared to IPs (more than 1 million domain) and the size of

the Drebin dataset. At this stage, we do not present the community detection and page ranking on the threat network; this will be conducted on a one-family use case in Section 5.3.4. ToGather leverages different malicious datasets, as previously described in Section 5.2.3, to tag the nodes of the produced threat network. Figure 5.2 depicts the diverse malicious activities of the nodes from Drebin threat network. First, the table shows the top PC malware families, which share network information with the Drebin threat network. For family names, we adopt the *Kaspersky* malware family naming as our ground truth. Besides, Figure 5.2 shows the percentage of each malicious type in the Drebin threat network. The result indicates that 56% of the shared nodes have a spamming activity, 40% are related to PC malware, 3% scanning, and 1% phishing activities. Notice that the previous percentages are only from the shared nodes and not from all the threat networks.



#	*Family*	*Hits*
1	Agent [a]	1268
2	VBNA	283
3	Adload	152
4	EgroupDial	121
5	TrustAsia	120
6	Vobfus	88
7	KuPlays	74
8	Pipibo	72
9	Sality	62

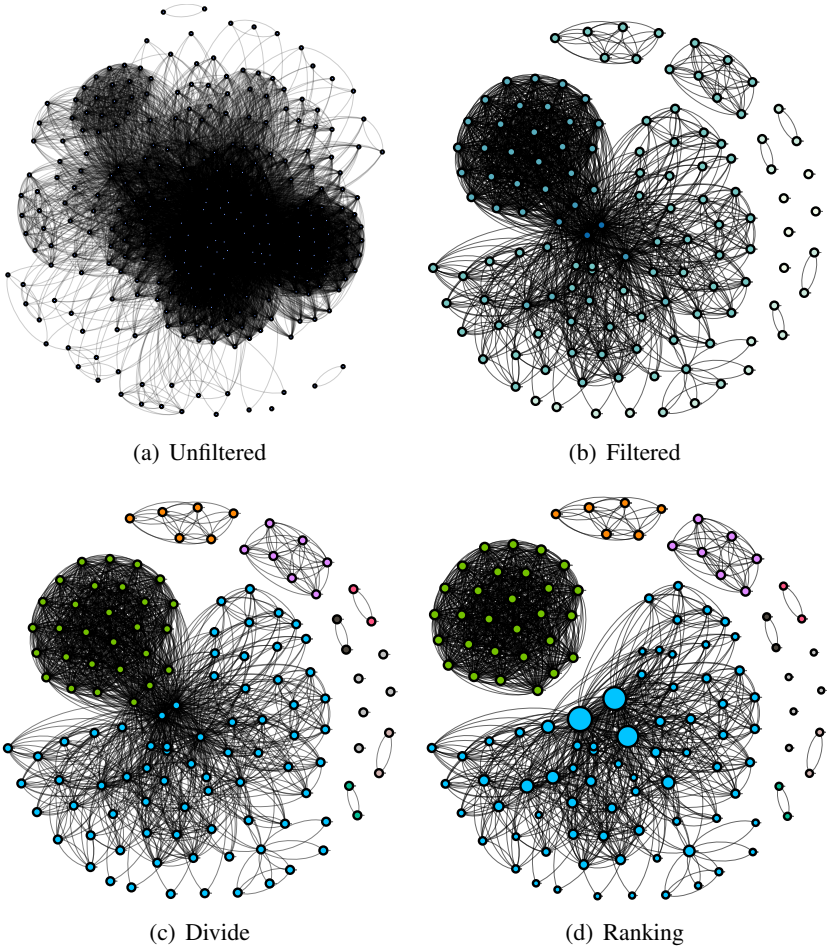
[a] Kaspersky Naming

Table 5.2: Drebin Dataset Tagging Results

5.3.4 Family Threat Networks

In this section, we present the results of ToGather in its typical usage scenario where malware samples from the same family are analyzed. Figure 5.8 shows the steps of generating threat networks from the `DroidKungFu` family sample. First, ToGather produces the threat network, including network information collected from the `DroidKungFu` analysis and Passive DNS correlation, as shown in Figure 5.8(a). Afterward, ToGather filters the whitelist network information. The results, shown in Figure 5.8(b), depict bright separated sub-threat networks without applying the community detection algorithm. This could be an insightful result for the security practitioner, especially that this sub-threat network contains network information exclusively from some samples. ToGather

goes a step ahead by applying both community detection (resolution hyperparameter $r = 3$) and page ranking algorithms (damping factor $d = 0.85$ and stopping criterion $\epsilon = 0.001$) as hyperparameters to divide the network and rank the importance of the nodes respectively. The result consists of multiple sub-threat networks, with high interconnection and low intra-connection, representing the cyber-infrastructures of `DroidKungFu` malware family.



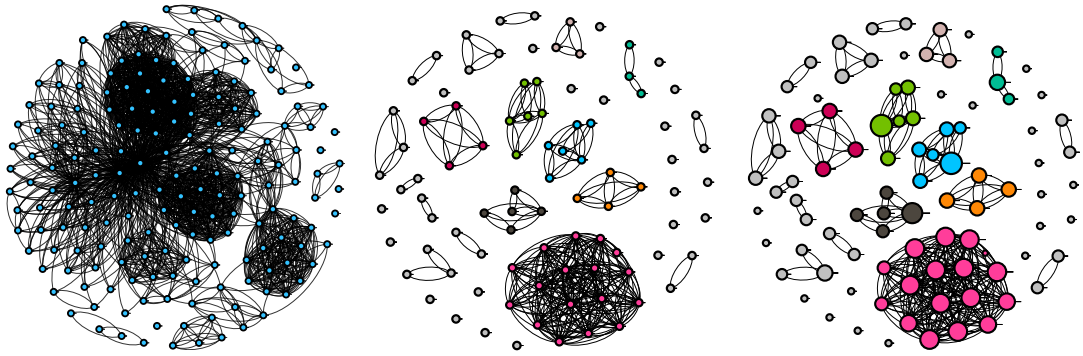
(a) Unfiltered (b) Filtered

(c) Divide (d) Ranking

Figure 5.8: DroidKungFu Malware Threat Network

Figure 5.9 shows the results provided by ToGather when using Android malware samples from `BaseBridge` family. Similarly, after the filtering operation, we could easily distinguish small sub-threat networks. In some cases, the community detection task could be optional due to the clear separation between the sub-threat networks. For instance, Figure 5.10 depicts the threat networks

for GinMaster, Adrd, and Plankton Android malware families before and after the community detection task. Here, Adrd family has multiple sub-threat networks without the need for the community detection function since it does not affect much the results. In the case of Plankton, it is necessary to detect and extract the sub-threat network.



(a) Unfiltered (b) Filter&Divide (c) Ranking

Figure 5.9: Basebrigde Malware Threat Network

Tables 5.3 and 5.4 show the top PC malware families and samples that share the network information with BaseBredge and DroidKungFu threat networks. An essential factor in the correlation is the explainability, where we could determine which network information is shared between the Android malware and PC malware. This could help the security investigator to track the other dimension of the adversary cyber-infrastructure.

#	*Sample*	*Hits*
1	ed7621ec4d [a]	2
2	e3bc76d14c	2
3	503902c503	1
4	bd9b87869b	1
5	8e0cf0a1ba6	1
6	f8a5cac12dc	1
7	14db95e5f6	1
8	9b5b576ef3	1
9	2ec2abc28d	1

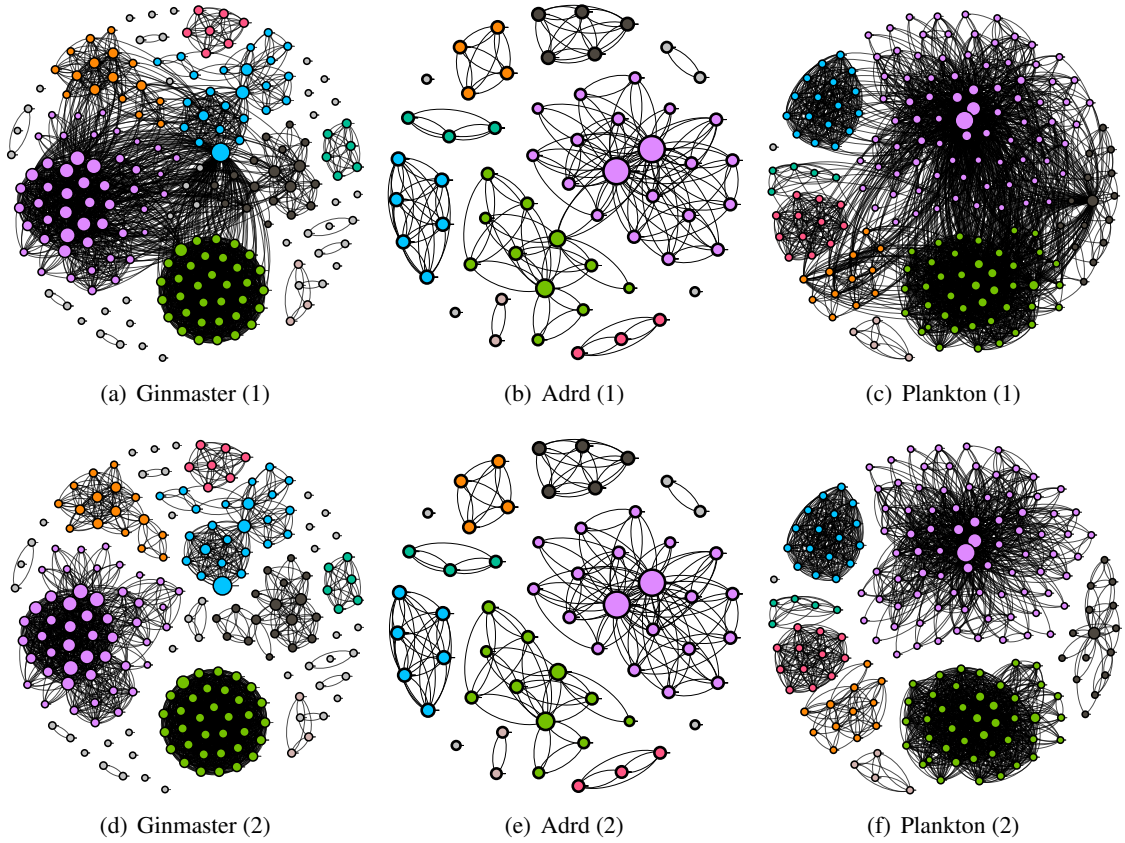
[a]MD5 Hash First 10 Chars

#	*Family*	*Hits*
1	Agent [a]	23
2	Vobfus	21
3	EgroupDial	13
4	Badur	9
5	LMN	7
6	WBNA	4
7	Pipibo	2
8	Blocker	2
9	Virut	2

[a]Kaspersky Naming

Table 5.3: Top PC Malware Related to BaseBridge Family

In addition to the PC malware tagging, we correlate with other cyber malicious activity datasets over the Internet. Figure 5.11 presents the malicious activities of DroidKungFu and BaseBridge families that are related to their threat network. Here, we find that both families could be part of a



(a) Ginmaster (1) (b) Adrd (1) (c) Plankton (1)

(d) Ginmaster (2) (e) Adrd (2) (f) Plankton (2)

Figure 5.10: Android Families From Drebin Dataset

spam campaign while having some scanning activity. Notice that these results represent a fraction of the actual activity because of limited datasets.

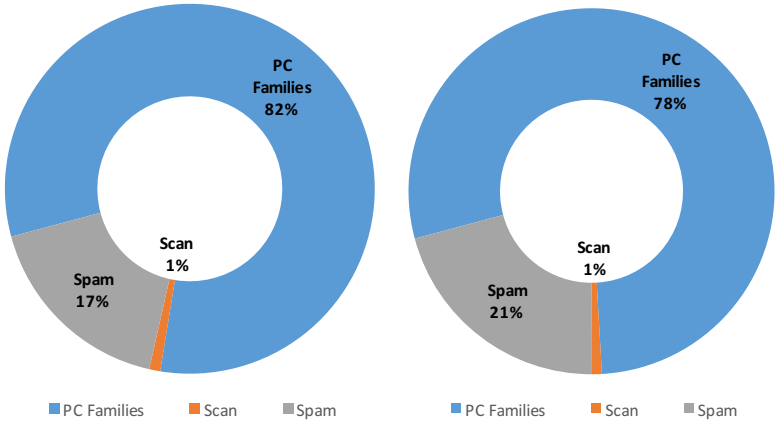
#	*Sample*	*Hits*
1	74529155cc [a]	3
2	bd5a9f768cf	2
3	259a244ab1	2
4	52da75225	1
5	11786afada	1
6	ad5e6d577b	1
7	9f4215bfc3	1
8	3c76ff67d0	1
9	117f21550	1

[a] MD5 Hash First 10 Chars

#	*Family*	*Hits*
1	Agent [a]	33
2	Adload	24
3	TrustAsia	13
4	KuPlays	11
5	Pipibo	8
6	FangPlay	5
7	StartPage	4
8	Injector	4
9	Turbobit	4

[a] Kaspersky Naming

Table 5.4: Top PC Malware Related to DroidKungFu Family



(a) BaseBridge (b) DroidKungFu

Figure 5.11: Maliciousness Tagging Per Family

5.4 Summary

In this chapter, we presented ToGather framework, a set of techniques, tools, and mechanisms as well as security feeds bundled together in order to achieve situational awareness about Android malware threats automatically. ToGather leverages state-of-art graph partitioning algorithms and multiple security feeds to produce insightful, granular, as well as actionable intelligence about malicious cyber-infrastructures related to Android malware samples. We evaluated ToGather on real malware from the Drebin Dataset [73]. The results show promising insights about cyber-infrastructures of Android malware families. The produced threat networks could show one side of the adversary infrastructure, which is the Android malware one; this side could lead to a larger threat network. Furthermore, all the results can be extracted automatically and periodically from a feed of Android malware samples belonging to one or various families. This requires fixing the hyperparameters related to the used algorithms of the community detection, and page ranking, as we did in our experimentation.

In this chapter as well as in Chapters 3 and 4, we propose Android malware fingerprinting systems that target the workstation category in the taxonomy proposed in Chapter 2. In the next chapter, we propose a portable Android malware detection system that targets all the deployment categories (mentioned in Chapter 2). More specifically, this involves a detection system that is

versatile enough to be efficiently deployed on high-end servers as well as on IoT devices such as Raspberry PI boards.

Chapter 6

Portable Supervised Malware Fingerprinting using Deep Learning

6.1 Overview

In this chapter, we propose MalDozer, an innovative and efficient framework for Android malware detection, leveraging sequence mining via neural networks. MalDozer focuses on portable malware detection based on applying supervised machine learning on static analysis features in contrast to Cypider, presented in Chapter 3, in which we propose an unsupervised system based on static analysis features. While Cypider provides a framework for malware clustering, aimed at market level app analysis. MalDozer provides an efficient malware detection to allow the deployement inside resource-constrained devices. MalDozer framework is based on an artificial neural network that takes, as input, the raw sequences of API method calls, as they appear in the DEX file, to enable malware detection and family attribution. MalDozer can automatically recognize malicious patterns using only the sequences of raw method calls in the assembly code.

6.1.1 Threat Model

We position MalDozer as an anti-malware system that detects Android malware and attributes it to a known family with high accuracy and minimal false positive and false negative rates. We assume that the analyzed Android apps, whether malicious or benign, are developed mainly in Java

or any other language that is translated to DEX bytecode. Therefore, Android apps developed by other means, e.g., web-based, are out of the scope of the current design of MalDozer. Also, we assume that apps' core functionalities are in DEX bytecode and not in C/C++ native code [19], i.e., the attacker is mainly using the DEX bytecode for the malicious payload. Furthermore, we assume that MalDozer detection results are not affected by malicious activities. In the case of a server, Android malicious apps are assumed to not modify the server system. However, in the case of deployment on infected mobiles or IoT devices, MalDozer should be protected from malicious activities to avoid tampering its results.

6.1.2 Usage Scenarios

The effectiveness of MalDozer, i.e., its high accuracy, makes it a suitable choice for malware detection in market level deployment (Taxonomy in Chapter 2), especially that its update only requires very minimal manual intervention. We only need to train MalDozer model on new samples without a *feature engineering*, since MalDozer can automatically extract and learn relevent malicious and benign features during the training. Notice that MalDozer could detect unknown malware based on our evaluation as presented in Section 6.3. Furthermore, due to the efficiency of MalDozer, it could be deployed on mobile devices such as phones and tablets. As for mobile devices, MalDozer acts as the detection component in the anti-malware app inside Android phones, where the goal is to check the maliciousness likelihood of new apps. Family attribution is very handy when detecting new malware apps. Indeed, MalDozer helps the anti-malware system to take the necessary precautions and actions based on the malware family, which could involve specific malicious threats such as ransomware.

6.2 Methodology

In this section, we present MalDozer framework and its components (Figure 6.1). MalDozer has a simple design, where a minimalistic preprocessing is employed to obtain the assembly code methods. As for the feature extraction (representation learning) and detection/attribution, they are

based on the actual neural network. This permits MalDozer to be very efficient with fast preprocessing and neural network execution. Since MalDozer is based on supervised machine learning, we first need to train our model. Afterward, we deploy the trained model along with a preprocessing procedure on the targeted devices. Notice that the preprocessing procedure is common between the training and the deployment phases to ensure the correctness of the detection results (Figure 6.1).

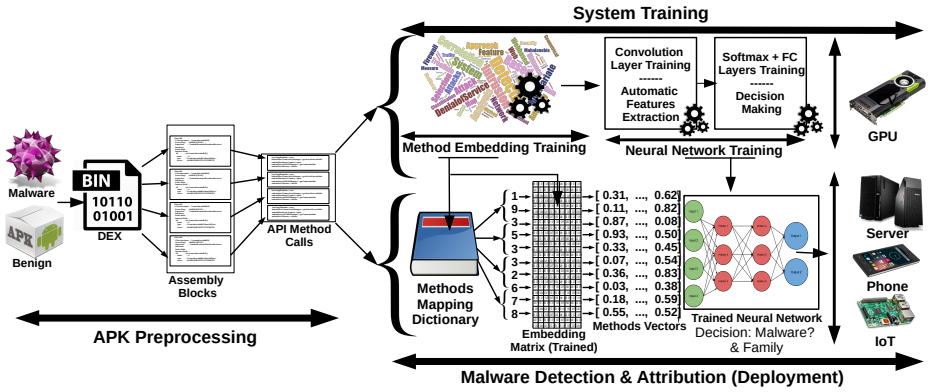


Figure 6.1: Approach Overview

1- Extraction of API Method Calls. MalDozer workflow extracts the sequences of API calls from Android app packages, in which we consider only *DEX* file. We disassemble the classes.dex to produce Dalvik VM assembly. Our goal is to model the Dalvik assembly to keep the maximum raw information with minimum noise. Notice here that we could use Android APIs (such as `android/net/ConnectivityManager` in Figure 6.2) instead of permission to have a granular view that helps distinguishing malware apps.

```
android/net/ConnectivityManager
android/net/ConnectivityManager
android/telephony/SmsManager
android/telephony/SmsManager
android/location/LocationManager
android/location/LocationManager
```

Figure 6.2: Android API from a Malware Sample

However, quantifying Android API could be challenging because there are plenty of common API calls shared between apps. Some solutions tend to filter only sensitive APIs and use them for detection. In this case, we require a manual categorization of sensitive APIs. Moreover, Android API

gives an abstract view of the actual malicious activity that could hinder malware detection. For this reason, we leverage Android API method calls as `android/net/ConnectivityManager;->getNetworkInfo` in Figure 6.3. By doing so, the proposed malware detector will have a more granular view of the app activity. In our case, we address this problem from another angle; we treat Android apps as a sequence of API method calls. We consider all the API calls with no filtering, where thier calling order is part of the information we use to identify malware. It represents the temporal relationship between two API method calls (in a given basic block) and defines the intended sub-tasks of the app. The sequence of API method calls preserves the temporal relationship over the individual basic blocks of the linear disassembly and ignores the order of the basic blocks. The obtained result is a merged sequence (Figure 6.1).

```
android/net/ConnectivityManager;->getNetworkInfo
android/net/ConnectivityManager;->getAllNetworkInfo
android/telephony/SmsManager;->sendTextMessage
android/telephony/SmsManager;->sendMultipartTextMessage
android/location/LocationManager;->getLastKnownLocation
android/location/LocationManager;->getBestProvider
```

Figure 6.3: Granular View Using API Method Calls

In other words, a *DEX* file, denoted by cd, is composed of a set of n compiled Java classes, $cd = \{cl_1, \cdots, cl_n\}$. Each Java class cl_i is, in turn, composed of a set of m methods, which are basic blocks, $cl_i = \{mt_1^i, \cdots, mt_m^i\}$. By going down to the API method level, mt_j^i is a sequence of k API method calls. Formally $mt_j^i = (P_1^{i,j}, \cdots, P_k^{i,j})$, where $P_l^{i,j}$ is the l^{th} API method call in method mt_j^i.

2- Dictionary Mapping of API Method Calls. In this step, we map the sequences of API method calls that are in an Android app to the corresponding identifiers. More precisely, we replace each API method with an identifier, resulting in a sequence of numbers. We also build a dictionary that maps each API call to its identifier. Notice that in the current implementation, the mapping dictionary is deployed with the learning model to map the API calls of the analyzed apps. In the deployment, we might find unknown API calls related to third party libraries. To overcome this problem: (i) We consider a large training dataset that covers most of the API calls. (ii) In the deployment phase, we replace unknown API calls with fixed identifiers. Afterward, we unify the

length of the sequences L (hyper-parameter) and pad a given sequence with zeros if its length $l < L$.

3- Unification of the Sequences' Size. The length of the sequences varies from one app to another. Hence, it is important to unify the length of the sequences. Two cases are depending on the length of the sequence and the hyper-parameter. We choose a uniform sequence size as follows: i) If the length of a given sequence is greater than the uniform sequence size L, we take only the first L items to represent the apps. ii) In case the length of the sequence is less than L, we pad the sequence with zeros. It is important to mention that the uniform sequence size hyper-parameter influences the accuracy of MalDozer. A simple rule is that the larger is the size, the better it is, but this will require a lot of computation power and a long time to train the neural network.

4- Generation of the Semantic Vectors. The identifier in the sequences needs to be shaped to fit as input to our neural network. The issue could be solved by representing each identifier by a vector. The question that arises is *how are such vectors produced?* A straightforward solution is to use one-hot vectors, where a vector has one in the interface value row, and zero in the rest. Such a vector is very sparse because its size is equal to the number of API calls, which makes it impractical and computationally prohibitive for the training and the deployment. To address this issue, we resort to dense vectors. These vectors are semantically related, and we could express their relation by computing a distance. The smaller the distance is, the more related the vectors are (i.e., API calls). We describe word embedding in Section 6.2.1. The output of this step is sequences of vectors for each app that keeps the order of the original API calls; each vector has a fixed size K (hyper-parameter).

5- Prediction using a Neural Network. The final component in MalDozer framework is the neural network, which is composed of several layers. The number of layers and the complexity of the model are hyper-parameters. However, we aim to keep the neural network model as simple as possible to reduce the execution time during its deployment, especially on IoT devices. In our design, we rely on the convolutional layers [138] to automatically discover the pattern in the raw method calls. The input to the neural network is a sequence of vectors, i.e., a matrix of $L \times K$ shape. In the training phase, we train the neural network parameters (layers weight) based on the

app vector sequence and its labels: (i) malware or benign for the detection task, and (ii) malware families for the attribution task. In the deployment phase, we extract the sequence of methods and use the embedding model to produce the vector sequence. Finally, the neural network takes the vector sequence to decide about the given Android app.

6.2.1 MalDozer Method Embedding

The neural network takes vectors as input. Therefore, we represent our Android API method calls as vectors. As a result, we formalize an Android app as a sequence of vectors with a fixed size (L). We could use one-hot vectors. However, their size is the number of unique API method calls in our dataset. This makes such a solution not scalable to a large-scale training. Also, the word embedding technique outperforms the results of the one-hot vector technique in our case [138, 153, 159]. Therefore, we seek a compact vector, which also has semantic value. To fulfill these requirements, we choose the word embedding techniques, namely, word2vec [153] and GloVe [159]. Our primary goal is to have for each Android API method a dense vector; the vector' values are learned from the method contexts in a large dataset of Android apps. Thus, in contrast to one-hot vectors, each word embedding vector contains a numerical summary of the Android API call semantic representation.

Moreover, the learned API call vectors have semantic relationships to each other in terms of functionality, i.e., developers tend to use specific API method calls in the same context. In our case, we learn these vectors from our dataset that contains benign and malicious apps by using word2vec [153]. The latter is a computationally efficient predictive model based on learning word embedding vectors, which are applied in our case to raw Android API method calls. The output obtained from training the embedding word model is a matrix $K \times A$, where K is the size of the embedding vector, and A is the number of unique Android API method calls. Both K and A are hyper-parameters; we use $K = 64$ in all our models. We choose $K = 64$ because: (i) it is a common practice in NLP literature to use $K \in \{32, 64, 128, ...\}$ values for word embedding vectors sizes. (ii) The value 64 is a good tradeoff between the efficiency, which is required in mobile deployment, and the effectiveness compared to the use of 32 and 128 embedding vector sizes. In the deployment phase (Figure 6.1), MalDozer uses the word embedding model and looks up for each API method call

identifier to find the corresponding embedding vector.

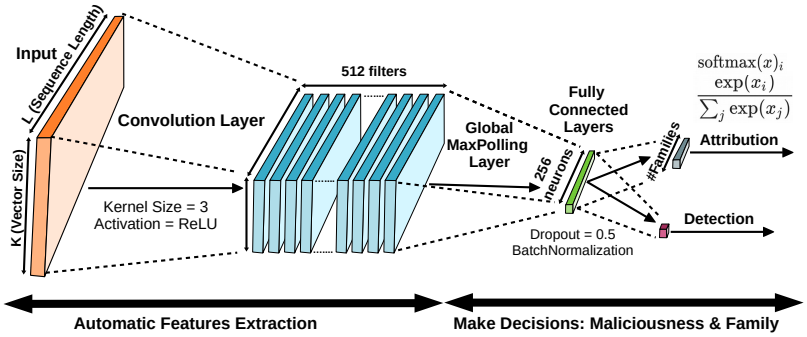


Figure 6.4: Neural Network Architecture

6.2.2 MalDozer Neural Network

MalDozer neural network is inspired by [138], where the authors use a neural network for sentence classification tasks such as sentiment analysis. The proposed architecture shows high results and outperforms many of state-of-the-art benchmarks with a relatively simple neural network design. We raise the questions: *Why could such a Natural Language Processing (NLP) model be useful in Android malware detection?* and *why do we choose to build it on top of this design [138]?* We formulate our answers as follows: i) NLP is a challenging field where we deal with text. So, there is an enormous number of vocabularies; also, we could express the same meaning in different ways. Besides, the same semantics could be expressed with different combinations of words, which is the equivalent of code obfuscation in natural language processing.

In our context, we deal with sequences of Android API method calls and want to find the combination of method calls patterns, which produces the same (malicious) activity. We use API method calls as they appear in the bytecode. Indeed, there is a temporal relationship between API methods in basic blocks. Nevertheless, the extraction process neglects the order among blocks and only considers the order inside the code blocks. By analogy to NLP, blocks are sentences, and the API method calls are words. Further, an app (paragraph) is a list of basic blocks (unordered sentences). Malware detection using API method calls looks easier compared to the NLP one because of the huge difference in the vocabulary, i.e., the number of Android API method calls is significantly less than the number of words in natural language. Also, combinations in natural language are much

more complex compared to Android API calls. ii) We choose to use this model due to its efficiency and ability to run on resource-constrained devices. Table 6.1 depicts the neural network architecture of MalDozer's detection and attribution tasks. Since both networks are very similar, the only notable difference is in the output layer.

In the detection task, we need only one neuron in the output layer because the network decides whether the app is malware or not. As for the attribution task, there are multiple neurons, one for each Android malware family. Having the same architecture for the detection and attribution makes the development and the evaluation of a given design simpler. Because the network architecture achieves good results in one task, it will have very similar results in the other one. As presented in Figure 6.4, the first layer is a convolution layer [138] with Rectified Linear Unit (ReLU) activation function ($f(x) = max(0, x)$). Afterward, we use the global max pool [138] and connect it to a fully-connected layer. Notice that in addition to Dropout [176] used to prevent over-fitting, we also utilize Batch Normalization [120] to improve our results. Finally, we have an output layer, where the number of neurons depends on the detection or attribution tasks.

#	*Layers*	*Options*	*Activ*
1	Convolution	Filter=512, FilterSize=3	ReLU
2	MaxPooling	/	/
3	FC	#Neurons=256, Dropout=0.5	ReLU
4	FC	#Neurons={1,#Families[1]}	Softmax

[1] The number of malware families in the training dataset.

Table 6.1: MalDozer Malware Neural Network

	Server (1/2)	**Laptop**	**RPI2**
GPU	TITAN X / no	no	no
CPU	Intel E5-2630	Intel T6400	ARM Core A7
RAM	128GB	3GB	1GB

Table 6.2: Hardware Specifications

6.2.3 Implementation

In this section, we present the software and hardware components of MalDozer implementation.

Software. We implement MalDozer using *Python* and *Bash* scripting languages. First, Python zip library extracts the *DEX* file from the *APK* file. We use *dexdump* command-line to produce

the assembly from DEX file. *Dexdump* is available through the Android SDK, but in the case of Raspberry PI, we build it from its source code. Regular expressions are employed to extract API method calls from the assembly. To develop the neural network, we use Tensorflow library [26]. Notice that there is no optimization in the preprocessing; in the runtime evaluation, we use only a single thread app.

Hardware. To evaluate the efficiency of MalDozer, we evaluate multiple types of hardware, as shown in Table 6.2, starting from servers to *Raspberry PI* [25]. For training, the Graphics Processing Unit (GPU) is a vital component because the neural network training needs immense computational power. The training takes hours under *NVIDIA TitanX*. However, the deployment could be virtually on any device, including IoT devices (such as Raspberry Pi). To this end, we consider *Raspberry PI* as an IoT device because it is one of the hardware platforms supported by Android Things [58]. We also use low-end laptops in our evaluation, as shown in Table 6.2.

6.3 Evaluation

In this section, we conduct our evaluation using different datasets that primarily cover the following performance aspects: (I) *Detection Performance*: We evaluate how effectively MalDozer can distinguish between malicious and benign apps in terms of F1-measure, precision, recall, and false-positive rate. (II) *Attribution Performance*: We evaluate how effectively MalDozer can correctly attribute a given malicious app to its malware family. (III) *Runtime Performance*: We measure the preprocessing and the detection runtime on different types of hardware.

6.3.1 Datasets

In our evaluation, we have two main tasks: i) Detection, which aims at checking if a given app is malware or not, ii) Attribution, which aims at determining the family of the detected malware. We conduct the evaluation experiments under two types of datasets: i) Mixed dataset, which contains malicious apps and benign apps, as presented in Table 6.3. ii) Malware dataset, which contains only malware, as shown in Table 6.4. As for the malware dataset, we leverage reference datasets such as

Malgenome [17] and *Drebin* [73]. We also collect two other datasets from different sources, e.g., *virusshare.com*, *Contagio Minidump* [21]. The total number of malware samples is $33K$, including Malgenome and Drebin datasets. As for the attribution task, we use only malware from the previous datasets, where each family has at least 40 samples, as presented in Tables 6.12, 6.13, and 6.14. To this end, we propose MalDozer dataset, as in Table 6.12, which contains 20K malware samples from 32 malware families. We envision to make MalDozer dataset available upon request for the research community. The benign app samples have been collected from *Playdrone* dataset [24]. We leverage the top $38K$ apps that are ranked by the number of downloads.

Dataset	#Malware	#Benign	Total
Malgenome	1,258	37,627	38.885
Drebin	5,555	37,627	43,182
MalDozer	20,089	37,627	57,716
All	33,066	37,627	70,693

Table 6.3: Datasets for Detection Task

Dataset	#Malware	#Family
Malgenome	985	9
Drebin	4,661	20
MalDozer	20,089	32

Table 6.4: Datasets for Attribution Task

6.3.2 Malware Detection Performance

We evaluate MalDozer on different cross-validation settings, two, three, five and ten-fold, to examine the detection performance under different training/test set percentages ($50\%, 66\%, 80\%, 90\%$) from the actual dataset (10 training epochs). Table 6.5 depicts the detection results on Malgenome dataset. MalDozer achieves excellent results, F1-Score=99.84%, with a small *False Positive Rate* (FPR), 0.04%, despite the unbalanced dataset, where benign app samples are the most dominant in the dataset. The detection results are similar under all cross-validation settings. Table 6.6 presents the detection results on Drebin dataset, which are very similar to the Malgenome ones. MalDozer reaches F1-Score=99.21%, with FPR=0.45%. Similar detection results are shown in Table 6.7 on MalDozer dataset (F1-Score=98.18% and FPR=1.15%). Table 6.8 shows the results related to all datasets, where MalDozer achieves a good result (F1-Score=96.33%). However, it has a higher false positive rate compared to the previous results (FPR=3.19%). This leads us to manually investigate the false positives. We discover, by correlating with *virusTotal.com*, that several false positive apps are already detected by many vendors as malware.

	F1%	P%	R%	FPR%
2-Fold	99.6600	99.6620	99.6656	0.06
3-Fold	98.1926	98.6673	97.9812	1.97
5-Fold	99.8044	99.8042	99.8045	0.09
10-Fold	**99.8482**	**99.8474**	**99.8482**	**0.04**

Table 6.5: Detection on Malgenome Dataset

	F1%	P%	R%	FPR%
2-Fold	98.8834	98.9015	98.9000	0.13
3-Fold	99.0142	99.0130	99.01579	0.51
5-Fold	99.1174	99.1173	99.1223	0.31
10-Fold	**99.2173**	**99.2173**	**99.2172**	**0.45**

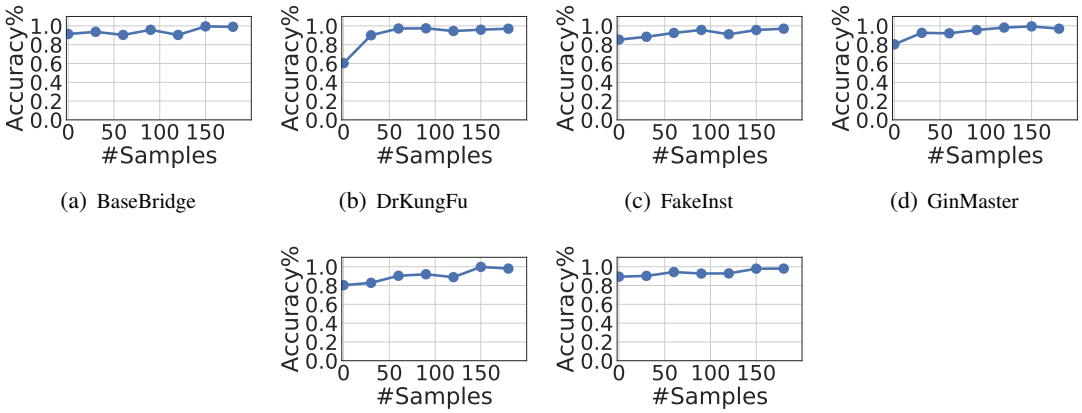
Table 6.6: Detection on Drebin Dataset

	F1%	P%	R%	FPR%
2-Fold	96.8576	96.9079	96.8778	1.01
3-Fold	97.6229	97.6260	97.6211	2.00
5-Fold	97.7804	97.7964	97.7753	2.25
10-Fold	**98.1875**	**98.1876**	**98.1894**	**1.15**

Table 6.7: Detection on MalDozer Dataset

	F1%	P%	R%	FPR%
2-Fold	96.0708	96.0962	96.0745	2.53
3-Fold	95.0252	95.0252	95.0278	4.01
5-Fold	96.3326	96.3434	96.3348	2.67
10-Fold	**96.2958**	**96.2969**	**96.2966**	**3.19**

Table 6.8: Detection on All Dataset



(a) BaseBridge (b) DrKungFu (c) FakeInst (d) GinMaster

(e) Opfake (f) Plankton

Figure 6.5: Evaluation of Unknown Malware Detection

Unknown Malware Detection

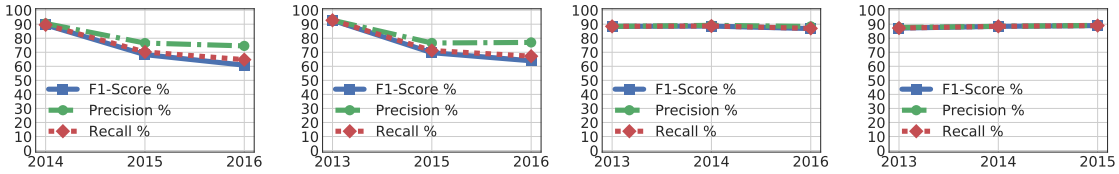
Although MalDozer demonstrates very good detection results, some questions still arise: (i) *Can MalDozer detect samples of unknown malware families?* and (ii) *How many samples are needed for a given family to achieve a good accuracy?* To answer these questions, we conduct the following experiment on Drebin mixed dataset (Malware + Benign), where we focus on top malware families, i.e., BaseBridge, DroidKungFu, FakeInstaller, GinMaster, Opfake, and Plankton. For each family, we train (5 epochs) our model on a subset dataset, which does not include samples of that family. These samples are used as a test set. Afterward, we train with few samples from the family and evaluate the model on the rest of the sample in that family. Progressively, we add more samples to the training and assess the accuracy of our model on detecting the rest of the

family samples. Answering the above questions: *(i) Can MalDozer detect unknown malware family samples?* Yes, Figure 6.5 shows the accuracy versus the number of samples in the training dataset. We see that MalDozer (zero samples versus accuracy) could detect the unknown malware family samples without previous training. The accuracy varies from 60% to 90%. *(ii) How many samples for a given family to achieve a good accuracy?* MalDozer needs only about 10 to 20 samples to reach 90% (Figure 6.5). In the case of DroidKungFu, MalDozer needs 20 samples to reach 90%. Considering only 10 to 20 samples from a malware family is a rather small number to obtain quality results. This varies from a malware family to another due to: (i) the similarity of the malicious pattern of the new family compared known patterns. The higher the similarity to existing malware, the better the detection performance with (sometimes without any family sample in the training dataset) minimum samples in the training dataset. (ii) Some new malware families tend to have simple patterns in their payload. Therefore, the learning system needs only a few samples to grasp the patterns of the whole family.

Resiliency Against API Evolution over Time

As we have seen in the previous section, MalDozer could detect new malware samples from unknown families using samples from *Drebin* dataset collected in the period of 2011/2012. We aim to answer another important question: *Can MalDozer detect malicious and benign apps collected in different years?* To answer this question, we evaluate MalDozer on four datasets collected from [67] spanning across four consecutive years: 2013, 2014, 2015, and 2016. We take five malicious apps and five benign apps from the samples of each year. Than, we train MalDozer in one year dataset and test it on the rest of the datasets. The obtained results show that MalDozer detection is more resilient to API evolution over time compare to the result reported in [148], as presented in Figure 6.6. Starting with 2013 dataset (Figure 6.6(a)), we train MalDozer on 2013 samples and evaluate it on 2014, 2015, and 2016 ones. We notice a high detection rate in the 2014 dataset since it is collected in the consecutive year of the training dataset. However, the detection rate decreases in 2015 and 2016 datasets, but it is above an acceptable detection rate (F1-Score=70%). Similarly, we obtained the results of the 2014 dataset, as depicted in Figure 6.6(b). Also, training MalDozer on 2015 or 2016 datasets exhibits excellent results under all the datasets collected in other years,

where we reach F1-Score from 90% to 92.5%.



(a) 2013 Dataset (b) 2014 Dataset (c) 2015 Dataset (d) 2016 Dataset

Figure 6.6: Detection versus Time

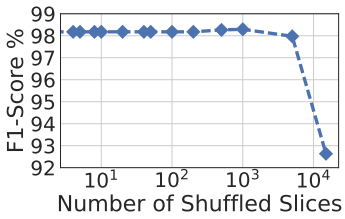


Figure 6.7: Shuffle Rate versus F1-Score

Resiliency against changing the order of API methods

In the following, we evaluate the robustness of MalDozer against changes in the order of API method calls. Such changes may occur for various reasons, such as: (i) We could use different disassembly tools in the production, (ii) A malware developer could repackage the same malicious app multiple times. The previous scenarios could lead to losing the temporal relations among the API calls. In the case of malware developer, she/he will be limited by keeping the same malicious semantics in the app. To validate the robustness of MalDozer against such methods that alter the order, we conduct the following experiment: First, we train our model on the training dataset. Afterward, we randomly shuffle the sequence of API method calls in the test dataset. We divide the testing app sequence into N blocks, then shuffle them and evaluate the F1-Score. We repeat until N is equal to the number of sequences, i.e., one API call in each block. The result of this experiment is shown in Figure 6.7. The latter depicts the F1-Score versus the number of blocks, starting with four blocks and ending with 15K blocks, where each block contains one API call. Figure 6.7 demonstrates the resiliency of MalDozer against changing the order of API method

calls. We observe that even with completely random individual API method calls, MalDozer still achieves 93% F1-score.

6.3.3 Family Attribution Performance

Family attribution is an important task for Android security, where MalDozer distinguishes itself from the existing malware detection solutions, since only few solutions provide this functionality. Starting with Malgenome dataset, MalDozer achieves a very good result, i.e., F1-Score of 99.18%. Similarly, MalDozer reaches an F1-Score of 98% on Drebin dataset. The results per malware family attribution performance for Malgenome and Drebin are presented in Tables 6.13 and 6.14. MalDozer achieves good results in the case of MalDozer dataset, F1-Score of 85%. Our interpretation of this result comes from Tables 6.12, 6.13 and 6.14, which depict the detailed results per malware family. For example, the family `agent` exhibts poor results because of the mislabeling, since `agent` is a common name for many Android malware families. We believe that there is a lot of noise in family labeling of the MalDozer dataset since we leverage only one security vendor for labeling. Despite this fact, MalDozer demonstrates acceptable results and robustness.

	F1%	P%	R%
2-Fold	98.9834	99.0009	98.9847
3-Fold	98.9910	99.0026	98.9847
5-Fold	99.0907	99.1032	99.0862
10-Fold	**99.1873**	**99.1873**	**99.1878**

Table 6.9: Attribution on Malgenome

	F1%	P%	R%
2-Fold	98.1192	98.1401	98.1334
3-Fold	98.6882	98.6998	98.6912
5-Fold	98.5824	98.5961	98.5839
10-Fold	**98.5198**	**98.5295**	**98.5196**

Table 6.10: Attribution on Drebin

	F1%	P%	R%
2-Fold	89.3331	89.5044	89.3424
3-Fold	81.8742	82.7565	81.8109
5-Fold	83.8518	84.1360	84.0061
10-Fold	**85.5233**	**85.6184**	**85.8479**

Table 6.11: Attribution on MalDozer

6.3.4 Run-Time Performance

In this section, we evaluate the efficiency of MalDozer, i.e., the runtime during the deployment phase. We divide the runtime into two parts: i) *Preprocessing time*: the required time to extract and preprocess the sequences of Android API method calls. ii) *Detection time:* time needed to predict a given sequence of API method calls. We measure the detection time according to the model complexity of different hardware. Figure 6.11(a) depicts the average preprocessing time, along with its standard deviation, related to each hardware. The server machines and the laptop spend, on average, 1 second in the preprocessing time, which is quite acceptable for production.

	Malware Family	#Sample	F1-Score
01	FakeInst	4822	96.15%
02	Dowgin	2248	84.24%
03	SmsPay	1544	81.61%
04	Adwo	1495	87.79%
05	SMSSend	1088	81.48%
06	Wapsx	833	78.85%
07	Plankton	817	94.18%
08	Agent	778	51.45%
09	SMSReg	687	80.61%
10	GingerMaster	533	76.39%
11	Kuguo	448	78.28%
12	HiddenAds	426	84.20%
13	Utchi	397	93.99%
14	Youmi	355	72.39%
15	Iop	344	93.09%
16	BaseBridge	341	90.50%
17	DroidKungFu	314	85.85%
18	SmsSpy	279	85.05%
19	FakeApp	278	93.99%
20	InfoStealer	253	82.82%
21	Kmin	222	91.03%
22	HiddenApp	214	76.71%
23	AppQuanta	202	99.26%
24	Dropper	195	77.11%
25	MobilePay	144	78.74%
26	FakeDoc	140	96.38%
27	Mseg	138	55.38%
28	SMSKey	130	81.03%
29	RATC	111	84.81%
30	Geinimi	106	95.58%
31	DDLight	104	90.55%
32	GingerBreak	103	84.87%

Table 6.12: MalDozer Android Malware Dataset

	Malware Family	#Sample	F1-Score
01	DroidKungFu3	309	99.83%
02	AnserverBot	187	99.19%
03	BaseBridge	121	98.37%
04	DroidKungFu4	96	99.88%
05	Geinimi	69	97.81%
06	Pjapps	58	95.65%
07	KMin	52	99.99%
08	GoldDream	47	99.96%
09	DroidDreamLight	46	99.99%

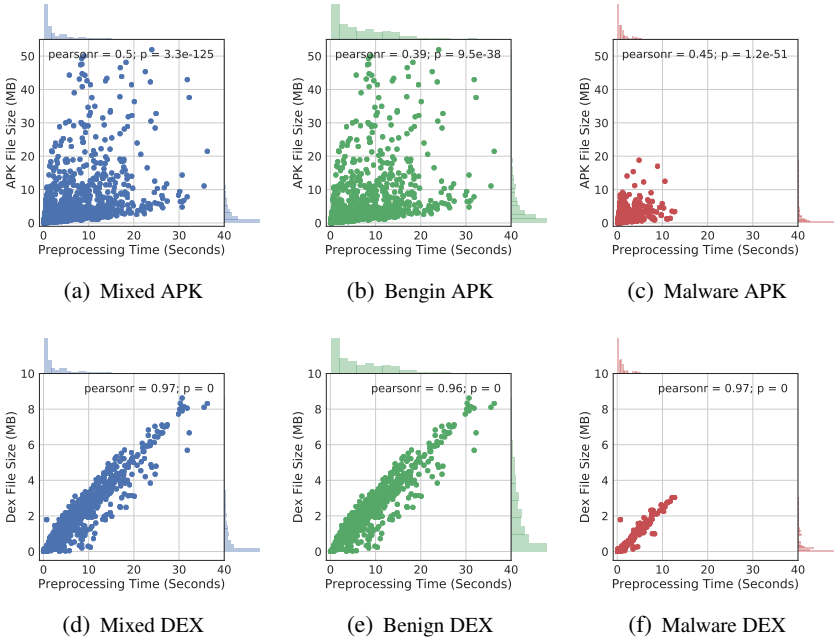
Table 6.13: Malgenome Attribution Dataset

	Malware Family	#Sample	F1-Score
01	FakeInstaller	925	99.51%
02	DroidKungFu	666	98.79%
03	Plankton	625	99.11%
04	Opfake	613	99.34%
05	GinMaster	339	97.92%
06	BaseBridge	329	97.56%
07	Iconosys	152	99.02%
08	Kmin	147	99.31%
09	FakeDoc	132	99.24%
10	Geinimi	92	97.26%
11	Adrd	91	96.13%
12	DroidDream	81	98.13%
13	Glodream	69	90.14%
14	MobileTx	69	91.97%
15	ExploitLinuxLotoor	69	99.97%
16	FakeRun	61	95.16%
17	SendPay	59	99.14%
18	Gappusin	58	97.43%
19	Imlog	43	98.85%
20	SMSreg	41	92.30%

Table 6.14: Drebin Attribution Dataset

Also, as mentioned previously, we do not optimize the current preprocessing workflow. On an IoT device [25], the preprocessing takes, on average, about 4 seconds, which is more than acceptable for such a small device. Figure 6.11(b) presents the detection time on average that is related to each hardware. First, it is noticeable that the standard deviation is quite negligible, i.e., the detection time is constant for all apps. Also, the detection time is very low for all the devices. As for the IoT device, the detection time is only 1.3 seconds. Therefore, the average time that MalDozer needs to decide for a given app is 5.3 seconds on average in case of an IoT device, as we know that preprocessing takes most of the time $(4/5.3)$. Here, we ask the following two questions: (i) *Which part in the preprocessing needs optimization?* (ii) *Does the preprocessing time depend on the size of APK or DEX file?* To answer these questions, we randomly select 1K benign apps and 1K malware apps. We measure the preprocessing time and correlate it with the size of APK and DEX files. Figure 6.8 shows the experimentation results in the case of an IoT device [25]. The scattered charts depict the

preprocessing time along with the size of the APK or DEX file for the mixed, the benign-only, and the malware-only datasets. From Figure 6.8, it is clear that the preprocessing time is linearly related to the size of the DEX file. We perform the same experiment on a server and a laptop, and we get very similar results, as shown in Figures 6.9 and 6.10. Finally, we notice that the size of benign apps tends to be bigger than the one of malicious apps. Thus, the preprocessing time of benign apps is longer.



(a) Mixed APK (b) Bengin APK (c) Malware APK

(d) Mixed DEX (e) Benign DEX (f) Malware DEX

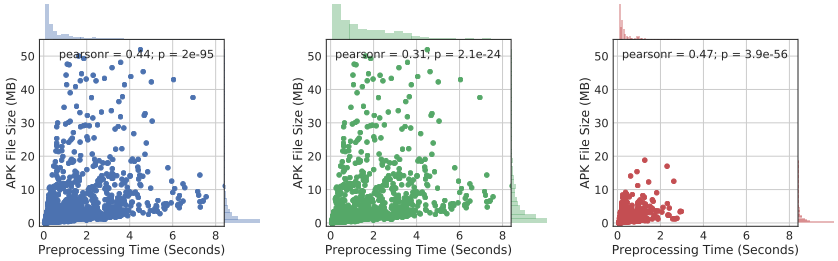
Figure 6.8: Preprocessing Time versus Package Sizes (IoT device)

	#Params	F1%	Word2Vec Size
Model 01	6.6 Million	98.95	100k
Model 02	4.6 Million	95.84	70k
Model 03	3.4 Million	93.81	50k
Model 04	1.5 Million	90.08	20k

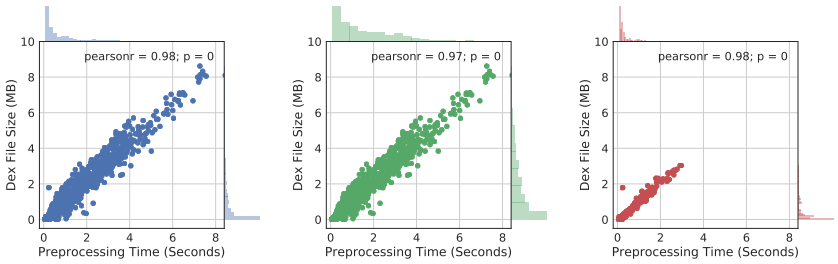
Table 6.15: Model Complexity versus Detection Performance

Model Complexity Evaluation

In this section, we examine the effect of model complexity on the detection time. By model complexity, we mean the number of parameters in the model, as depicted in Table 6.15. Many hyper-parameters can influence the complex nature of the model, but we primarily consider the

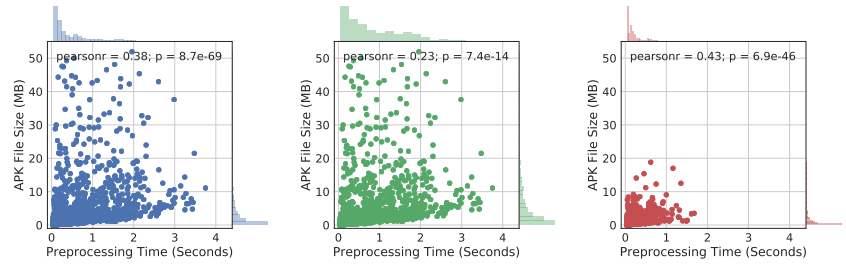


(a) Mixed APK (b) Bengin APK (c) Malware APK

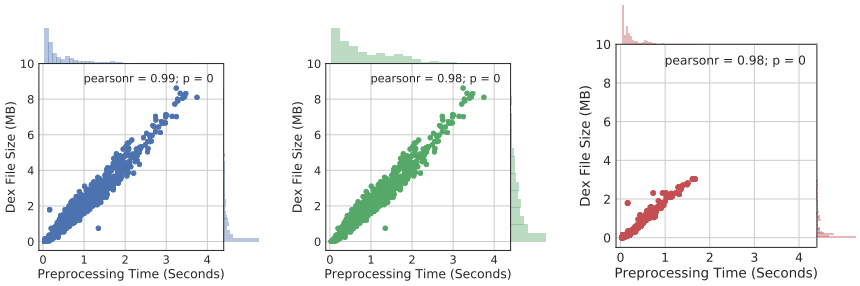


(d) Mixed Dex (e) Bengin Dex (f) Malware Dex

Figure 6.9: Preprocessing Time versus Package Sizes (Laptop)

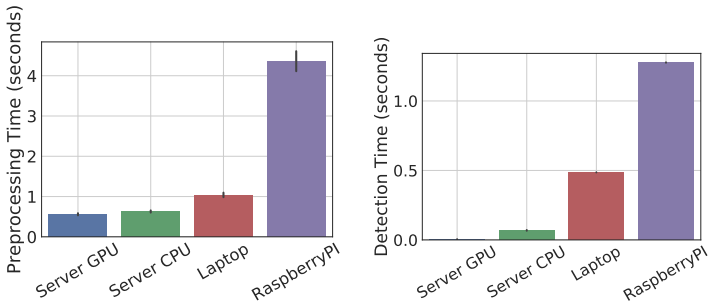


(a) Mixed APK (b) Bengin APK (c) Malware APK



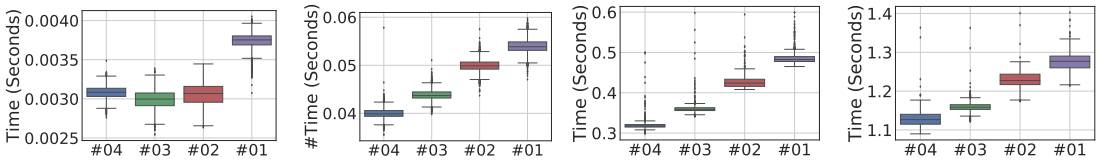
(d) Mixed Dex (e) Bengin Dex (f) Malware Dex

Figure 6.10: Preprocessing Time versus Package Sizes (Server)



(a) Preprocess (b) Prediction

Figure 6.11: Run-Time versus Hardware



(a) Server GPU (b) Server CPU (c) Laptop (d) IoT

Figure 6.12: Detection Time versus Model Complexity

word2vec embedding size. The latter is crucial for the detection of the model, especially if we have a big dataset. Table 6.15 demonstrates the complexity of the model versus the F1-Score. It is noticeable that the larger the number of parameters is, the more its performance increases. Based on our observation, bigger models are more accurate and more robust to changes. Finally, Figure 6.12 displays the execution time of the models in Table 6.15 on the IoT device. The detailed execution related to all the hardware is presented in Figure 6.12.

6.4 Summary

In this chapter, we presented MalDozer, an automatic, efficient, and effective Android malware detection, and attribution system. MalDozer relies on deep learning techniques and raw sequences of API method calls to identify Android malware. We have evaluated MalDozer on several small and large datasets, including *Malgenome*, *Drebin*, and our MalDozer dataset, in addition to a dataset of benign apps downloaded from Google Play. The evaluation results show that MalDozer is highly

accurate in terms of a malware detection as well as their attribution to corresponding families. Moreover, MalDozer can efficiently run under multiple deployment architectures, ranging from servers to small IoT devices. This work represents a step towards practical, automatic, and effective Android malware detection and family attribution.

In the next chapter, we propose a system that focuses on the robustness and the adaptability aspects in Android malware detection. First, the detection system should show resiliency to common code obfuscation and transformation techniques. Second, the detection performance should be resilient to the operating system, and malware changes overtime by employing an adaptation mechanism to handle changes.

Chapter 7

Resilient and Adaptive Android Malware Fingerprinting and Detection

7.1 Overview

In this chapter, we present PetaDroid, an Android detection system that provides, in contrast to MalDozer (previous chapter), (i) resiliency to common obfuscation techniques by introducing code randomization during training; (ii) adaptation to operating system and malware change overtime by introducing the use of confidence-based decisions to collect adaptation datasets overtime. In this context, we identify several limitations and gaps in the state-of-the-art Android malware detection solutions.

First, the accuracy of Android malware detection systems tends to decrease over time due to different factors (new OS versions, new malware families, new malicious techniques). We use time resiliency terminology to denote the adaptation of the Android malware detection solution to the change of malware, its attack techniques, and the Android platform. (i) It is important to note that detection systems may not be able to detect recently discovered samples because the system did not see such family samples during the training phase. However, the changes introduced by new families are, in most cases, incremental compared to existing malware threats. (ii) Malware developers build attacks against Android devices exploiting these zero-day flaws. The developed malware samples may be part of an existing family, but such samples could also exhibit some

variation in their implementation, depending on the employed the exploitation techniques. Such variant malware indicate a progressive evolution that could deceive detection systems over time. (iii) New APIs are introduced in the Android platform in every new version or update to the operating system and its ecosystem services. New APIs provide new capabilities for developers to build new app functionalities. On the other hand, malware developers could exploit these APIs to make new malicious functionalities to deceive existing Android malware detection. In time, the problem accumulates from one year to another, as seen in previous solutions [148, 158]. The gap between the detection capability and the emergence of new threats is indeed increasing, but the effect is minimum on a given time and incremental, which could be addressed timely. Therefore, we say that an Android malware detection solution is time-resilient if it can adapt to the changes of the Android platform as well as the changes in benign and malicious app patterns. A crucial component to adaptation is the ability of the solution to generalize from a small number of samples. Therefore, the detection performance on a small training set is an essential requirement for modern solutions. Because the sooner our detection solution can grasp the malicious patterns from a small dataset, the faster we can detect such threat, especially at a market level.

Second, only a few existing solutions, [73, 132], provide Android malware family attribution functionality. Furthermore, these solutions are built using supervised learning where prior knowledge on the families is required. However, such knowledge is hard to get and not realistic in many cases, especially for new families.

Third, malware developers employ various obfuscation techniques to thwart detection attempts. Obfuscation resiliency is a key requirement in modern malware fingerprinting that employs static analyses. Very few solutions address the obfuscation issue in the context of Android malware detection. Existing obfuscation resilient solutions such as DroidSheive [177] require manual feature extraction.

Forth, solutions, such as DroidSheive [177], StormDroid [91], and Drebin [73], rely on manual feature engineering based on classification techniques such as SVM and KNN. Despite their good detection performance, their approach is not scalable to the amount and the growth pace of Android malware. Therefore, there is an increasing need for solutions that are based on automatic feature engineering using deep learning and NLP techniques.

Fifth, state-of-the-art solutions, such as MaMaDroid [148, 158], require a lot of computing power due to the complex preprocessing, which affects the overall efficiency. The last aspect is an important requirement for Android malware detection due to the growing number of Android apps.

PetaDroid aims at satisfying all the aforementioned issues/requirements of modern Android malware fingerprinting by addressing the previously identified gaps and challenges in existing state-of-the-art solutions.

7.2 Methodology

In this section, we detail PetaDroid methodology and its components.

7.2.1 Approach

PetaDroid employs static analysis techniques on Android Dalvik Virtual Machine (DVM) binary code (DEX) to check the maliciousness of Android apps. PetaDroid starts by extracting raw static features from the Android Packaging (APK), specifically the Dalvik VM bytecode (DEX). We develop a fast preprocessing phase to extract raw Dalvik assembly instructions. We generate, on the fly, the canonical form of the assembly instructions by substituting the value of constants, memory addresses with symbolic names. The output is a raw sequence of canonical assembly instructions of Dalvik virtual machine.

PetaDroid maintains a logical separation among the software component of applications. We keep track of classes and methods' instruction sequences within the application global instruction sequence. It is a natural breakdown because an Android app is a set of classes, and a class is a set of methods and attributes. The global app execution sequence is composed of a list of micro-execution paths (method sequences), through which the execution proceeds during runtime. The extracted canonical instruction sequences help preserving the underlying micro-execution paths of the app while without and emphasison the global execution order. Micro-execution paths are instruction and API sequences of code functions (classes' methods).

Previous solutions [158] apply heavy and complex preprocessing to construct a global call graph

to simulate runtime execution. In contrast, our extraction approach is lightweight (very little computation needed in the preprocessing) because we consider only the method order within each given class methods. We argue that tracking the method order is sufficient to identify malicious apps. It allows swift preprocessing in commodity hardware while maintaining the intended granularity. Furthermore, and in contrast with previous solutions [158], we adopt granular features using a canonical instruction flowed by representation learning. We propose custom code modeling techniques for representation learning inspired by advanced natural language processing techniques. Specifically, we design and develop *Inst2Vec* and *InstNGram2Bag* code modeling techniques to model and discover latent information to produce embeddings from canonical instruction sequences.

In a nutshell, PetaDroid has six phases:

(1) **Representation Learning**: PetaDroid learns latent representations using unsupervised word2vec techniques [153].

(2) **Malware Detection**: PetaDroid employs neural network module as features' selector from the embedding representation. In the classification task, the training dataset guides feature selection to make the right detection outcome. PetaDroid classification system rests on ensemble deep learning models (CNN) that consume *Inst2Vec* embedding features to fingerprint malicious applications.

(3) **Detection Adaptation**: PetaDroid classification ensemble produces a detection confidence probability. Apps detected with high confidence, whether malicious or benign, will extend PetaDroid primary labeled dataset (used to build the current detection ensemble). Periodically, PetaDroid makes a new detection ensemble on the new dataset (primary and extended).

(4) **Code Representation**: For the detected malware, we produce feature vectors using N-grams bag of words and feature hashing techniques on top of the canonical instruction sequence. The outcome is what we call an InstNGram2Bag vector for each detected malware. An InstNGram2Bag vector summarize the intrinsic semantic of Android malware.

(5) **Digest Generation**: we produce digests by applying deep neural auto-encoders [116] on the InstNGram2Bag vectors to produce a compact embedding or a digest for each malicious sample.

(6) **Malware Family Clustering**: PetaDroid clusters the flagged malicious apps into groups with high inter-similarity between their digests, and most likely of the same malware family. PetaDroid

clustering system is based on DBScan[1] clustering algorithm.

7.2.2 Android App Representation

In this section, we present the preprocessing of Dalvik code and its representation into a sequence of canonical instructions. We seek the preservation of the maximum information about apps' behaviors while keeping the process very efficient. The preprocessing begins with the disassembly of an app bytecode to Dalvik assembly code, as depicted in Figure 7.1.

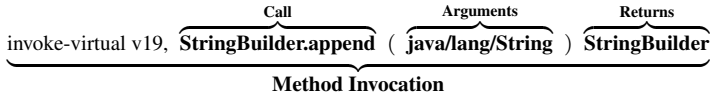
```
// Object Creation
new-instance v10, java/util/HashMap
// Object Access
invoke-direct v10, java/util/HashMap
if-eqz v9, 003e
..
// Method Invocation
// * = Android/telephony
invoke-virtual v4, */TelephonyManager.getDeviceId()java/lang/String
move-result-object v11
// Method Invocation
invoke-virtual v4, */TelephonyManager.getSimSerialNumber()java/lang/String
move-result-object v13
// Method Invocation
invoke-virtual v4 */TelephonyManager.getLine1Number()java/lang/String
move-result-object v4
...
// Object Creation
new-instance v20, java/io/FileReader
const-string v21, "/proc/cpuinfo"
invoke-direct/range v20, v21, java/io/FileReader.init(java/lang/String)
new-instance v21, java/io/BufferedReader
...
move/from16 v2, v20
// Field Access
// * = Android/content/pm
iget-object v0, v0, */ApplicationInfo.metaData Android/os/Bundle
move-object/from16 v19, v0
```

Figure 7.1: Android Assembly from a Malware Sample

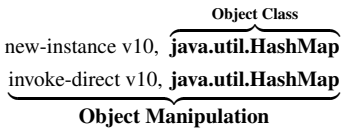
We model the Dalvik VM assembly code as code fragments where each fragment is a method code in the Dalvik assembly. It is a natural separation because Dalvik code D is composed of a set of classes $D = \{C_1, C_2, \ldots, C_s\}$. Each class C_i contains a set of methods $C = \{M_1, M_2, \ldots, M_k\}$ where we find actual assembly code instructions. We preserve the order of Dalvik assembly instructions within methods while ignoring the global execution paths. Method execution is a possible *micro-behavior* for an Android app, while a global execution path is a likely *macro-behavior*. PetaDroid assembly preprocessing produces a list of instruction sequences $P = \{S_1, S_2, \ldots, S_h\}$

[1] https://en.wikipedia.org/wiki/DBSCAN

where each sequence S contains an ordered instruction $S = \langle I_1, I_2, \ldots, I_v \rangle$. Thus, a sequence S defines a possible micro-execution (or behavior) from the Android app's overall runtime execution.



(12)



(13)

(14)

Figure 7.2: Canonical Representation of Dalvik Assembly

As shown in Figure 7.1, Dalvik assembly is too sparse. We want to keep the assembly instruction skeleton that reflects possible runtime behaviors with less sparsity. In PetaDroid (in contrast to MalDozer in previous chapter), we propose a canonical representation for Dalvik assembly code as shown in Figure 7.2. The key idea is to keep track of the Android platform APIs and objects utilized inside the method assembly. In order to fingerprint malicious apps, the canonical representation will mostly preserve the actions and the manipulated system objects, such as sending SMS action or getting (setting) sensitive information objects. PetaDroid canonical representation covers three types of Dalvik assembly instructions namely: *Method invocation*, *object manipulation*, and *field access*, as shown in Figure 7.2. In the method invocation, we focus on the method call, *Package.ClassName.MethodName*, the parameters list, *Package.ClassName*, and the return type, *Package.ClassName*. In object manipulation, we capture the class object, *Package.ClassName*, that is being used. Finally, we track the access to system fields by capturing the field name, *Package.ClassName.FieldName*, and its type, *Package.ClassName*. Our manual inspections of Dalvik assembly for hundreds of malicious and benign samples shows that these three forms cover the essential of Dalvik assembly instructions.

PetaDroid instruction parser keeps only the canonical representation and ignores the rest. For example, our experiments show that Dalvik opcodes add a lot of sparsity without enhancing the malware fingerprinting performance. On the contrary, it could affect the overall performance [152]

```
java/util/HashMap
java/util/HashMap
..
Android/telephony/TelephonyManager.getDeviceId()
java/lang/String
Android/telephony/TelephonyManager.getSimSerialNumber()
java/lang/String
Android/telephony/TelephonyManager.getLine1Number()
java/lang/String
...
java/io/FileReader
java/io/FileReader.init()
java/lang/String
java/io/BufferedReader
...
Android/content/pm/ApplicationInfo.metaData
Android/os/Bundle
```

Figure 7.3: Flatten Canonical Representation form a Malware Sample

negatively. The final step in the preprocessing of a method M (see Figure 7.1) is to flatten the canonical representation of a method into a single sequence S (see Figure 7.3). In the current design, we keep only Android platform related assets like API, classes, and system fields in the final method's sequence S. For this purpose, we maintain a vocabulary dictionary (assets names of Android platform) $V = \{\langle Asset_1, 1\rangle, \ldots, \langle Asset_d, d\rangle\}$ to filter and discretize the method sequence during the preprocessing. The output of the app representation phase is a list of sequences $P = \{cS_1, cS_2, \ldots, cS_h\}$. Each sequence is an ordered canonical instruction representation of one method.

7.2.3 Malware Detection

In this section, we present PetaDroid malware detection process using CNN on top of *Inst2Vec* embedding features. The detection process starts from a list of discretized canonical instruction sequences $P = \{cS_1, cS_2, \ldots, cS_h\}$. PetaDroid CNN ensemble produces a detection result together with maliciousness and benign detection probabilities for a given sample. In order to achieve automatic adaptation over time, we leverage the detection probabilities to automatically collect an extension dataset that PetaDroid employs to build new CNN ensemble models.

Fragment Detection

The fragment-based detection is a key technique in PetaDroid Android malware fingerprinting. A fragment F is a truncated portion from the concatenation cP of $P = \{cS_1, cS_2, \ldots, cS_h\}$. The size $|F|$ is the number of canonical instructions in the fragment F and it is a hyper parameter in PetaDroid framework. For a sequence cS_i, the order of canonical instructions is preserved within a method. In other words, we grantee the preservation of order inside the method sequence or what we refer to as a *micro-action*. On the other hand, no specific order is assumed between methods' sequences or what we refer to as *macro-action* (or behavior). In our context, and before we truncate cP into size $|F|$, we propose applying a random permutation on P to produce a random order in the macro-behavior while preserving its methods' micro-behaviors. The randomization happens in every access, whether it is during training or deployment phases. Each Android sample has $\frac{h!}{(h-k)!}$ possible permutation for the methods' sequences $P = \{cS_1, cS_2, \ldots, cS_h\}$, where h is the number of methods' sequence in a given Android app; and k is number of sampled sequences. Notice that the size of the concatenated k sequences must be greater than $|F|$ (fragment size hyperparameter).

The intuition behind fragment detection is the abstraction of Android apps behavior into a list of very small *micro-actions*. We consider each method canonical instruction sequence cS as possible *micro-actions* for an Android app. In a fragment, we keep the possible micro-actions intact and discard the app flow graph. We argue that this will force pattern learning, during the training, to focus on only micro-actions, which allows better generalization. Fragment-based detection has many advantages in the context of malware detection. First, fragment detection plays the role of dataset augmenter, which allows the learning model to generalize better from a small dataset. Second, it challenges the machine learning model and its training process to learn dynamic patterns at every training epoch. In other words, it focuses the model on robust, distinctive patterns from a sample of random micro-actions of methods. Third, we argue that our fragment-based detection helps improving the robustness of the malware detection model against conventional obfuscation techniques and code transformation in general. Fourth, in the testing phase, PetaDroid infers the maliciousness of a given sample by applying PetaDroid CNN on multiple sample fragments to obtain a detection decision with a specific confidence interval.

Inst2Vec Embedding

Inst2Vec is based on *word2vec* [153] technique to produce an embedding vector for each canonical instruction in our sequences. *Inst2Vec* is trained on instruction sequences to learn instructions semantics from the underlying contexts. This means that *Inst2Vec* learns a dense representation of a cononical instruction that reflect the instruction co-occurrence and context. The produced embeddings capture the semantics of instructions and translate into geometric values over multiple dimensions. Our Android malware detection technique is inspired by word2vec, modern NLP techniques as well as neural machine translation techniques. Furthermore, embedding features show high code fingerprinting accuracy and resiliency to common obfuscation techniques [100]. Word2vec [153] is a vector space model to represent the words of a document in a continuous vector space where words with similar semantics are mapped closely in the space. From a security perspective, we want to map our features (canonical instructions in a fragment) to continuous vectors where their semantics is translated to a distance in the vector space. Word2vec is a neural probabilistic model that is trained using the maximum likelihood concept. More precisely, given sequence of words: $w_1, w_2, \cdots, w_T$, at each position $t = 1, \cdots, T$, the model predicts a context of sequence within a window of fixed size m given center word w_j (illustrated in Equation 15), where m is the size of the training context [153].

$$L(\theta) = \prod_{t=1}^{T} \prod_{-m \leq j \leq +m, j \neq 0} P(w_{t+j}|w_t; \theta) \tag{15}$$

The objective function [153] $J(\theta)$ is the negative log likelihood as shown in Equation 17. The probability $P(w_{t+j}|w_t; \theta)$ is defined in Equation 19, where v_w and v'_w are the input and the output of the embeddings of w.

$$J(\theta) = -\frac{1}{T} \log L(\theta) \tag{16}$$

$$= -\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \leq j \leq m, j \neq 0} \log P(w_{t+j}|w_t; \theta) \tag{17}$$

$$P(w_O|w_I) = \text{softmax}(\grave{v}_{w_O}^T v_{w_I}) \tag{18}$$

$$= \frac{\exp(\grave{v}_{w_O}^T v_{w_I})}{\sum_{w=1}^{W} \exp(\grave{v}_w^T v_{w_I})} \tag{19}$$

We train the embedding model by maximizing log-likelihood as illustrated in Equation 21.

$$J_{\text{ML}} = \log P(w_O|w_I) \tag{20}$$

$$= (\grave{v}_{w_O}^T v_{w_I}) - \log \left(\sum_{w=1}^{W} \exp\{\grave{v}_w^T v_{w_I}\} \right). \tag{21}$$

Classification Model

Our single CNN model takes *Inst2Vec* features, which are a sequence of embeddings, each embedding captures the semantics of instructions. The temporal convolutional neural network [138], or 1-Dimensional CNN [198], is the working core component in PetaDroid single classification model. Table 7.1 details the architecture of our CNN single model.

#	***Layers***	***Options***
1	1D-Conv	Filter=128, Kernel=(5,5), Stride=(1,1), Padding=0, Activation=ReLU
2	BNorm	BatchNormalization
3	Global Max Pooling	/
4	Linear	#Output=512 , Activation=ReLU
5	Linear	#Output=256 , Activation=ReLU
6	Linear	#Output=1 , Activation=ReLU

Table 7.1: PetaDroid CNN Detection Model

The non-linearity used in our model employ the rectified linear unit (ReLUs) $h(x) = \max\{0, x\}$. We used Adam [113] optimization algorithm with a mini-batch of size fo 32 and a learning rate $3e-4$ for 100 epochs in all our experiments.

Dataset Notation

In this section, we present the notations that will be used in the next sections.

$X = \{(\langle cP_0, y_0\rangle, \langle cP_1, y_1\rangle, .., \langle cP_m, y_m\rangle\}$: X is the global dataset used to build ensemble models and report PetaDroid performance on various tasks. Where m is number of $\langle sample, label\rangle$ records in the global dataset X.

$X = \{X_{build}, X_{test}\}$: We use a build set X_{build} to train and tune the hyper-parameters of PetaDroid models. The test set X_{test} represents Android apps that the system will receive during the deployment. The test set X_{test} is used to measure the final performance of PetaDroid, which is reported in the evaluation section. X is split randomly into X_{build} (50%) and X_{test} (50%).

$X_{build} = \{X_{train}, X_{valid}\}$: The build set, X_{build}, is composed of a training set X_{train} and a validation set X_{valid}. It is used to build PetaDroid single CNN models for the CNN ensemble. For each single CNN model, we tune the model parameters to achieve the best detection performance on X_{valid}. The build set $m_{build} = m_{train} + m_{valid}$: is the total number of records used to build PetaDroid. The training set takes 80% of the build set X_{build} and 20% of X_{build} is used for the validation set X_{valid}.

Detection Ensemble

PetaDroid detection component relies on an ensemble $\Phi = \langle sC_1, sC_2, \ldots, sC_\phi\rangle$. Ensemble Φ is composed of ϕ single CNN models The number of single CNN models in the ensemble ϕ is a hyperparameter. We fixed $\phi = 6$ in the evaluation experiments. As mentioned previously PetaDroid trains each CNN model C for number of epochs ($epochs = 100$). In each epoch, we compute $Loss_T$ and $Loss_V$, the *training* and *validation* losses respectively, and save a snapshot of the single CNN model parameters. $Loss_T$ and $Loss_V$ are the log loss across training and validation sets:

$$p = singleCNN_\theta(y = 1|cP)$$

$$loss(y, p) = -(y\log(p) + (1 - y)\log(1 - p)),$$

$$Loss_T = \frac{-1}{m_{train}} \sum_{i=1}^{m_{train}} loss(y_i, p_i),$$

$$Loss_V = \frac{-1}{m_{valid}} \sum_{i=1}^{m_{valid}} loss(y_i, p_i),$$

Where p is the maliciousness likelihood probability given a fragment F (a concatenated and truncated canonical instructions cP) and model parameters θ (Section 7.2.2). PetaDroid selects automatically the top ϕ models from the saved model snapshots that have the lowest *training* and *validation* losses $Loss_T$ and $Loss_R$ respectively.

$$\hat{y} = \Phi(x) \quad = \frac{1}{\phi}\left(\sum_{i}^{\phi} sC_i(x)\right) \tag{22}$$

PetaDroid CNN ensemble Φ produces a maliciousness probability likelihood by averaging the likelihood probabilities of single CNN models sC, as shown in Equation 22.

Confidence Analysis

PetaDroid ensemble computes the maliciousness probability likelihood $Prob_{Mal}$ given a fragment F, as follows:

$$\hat{y} = \Phi(F), \;\; Prob_{Mal} = \hat{y}, \;\; Prob_{Ben} = (1 - \hat{y})$$

Previous Android malware detection solutions, such as [91, 132, 148], utilize a simple detection technique (we refer to it as a *general decision*) to decide on the maliciousness of Android apps. In the *general decision*, we compute general threshold $\zeta \in [0, 1]$ that achieves the highest detection performance on the validation dataset X_{valid}. In the deployment phase (or evaluation in our case on X_{test}), The general decision D_ζ utilize the computed threshold ζ to make detection decisions:

$$D_\zeta = \begin{cases} Malware & Prob_{Mal} > \zeta \\ Benign & Prob_{Mal} <= \zeta \end{cases}$$

PetaDroid employs F1-score as detection performance metric to automatically select ζ and to report general detection performance on the test set X_{test} during our evaluation, in Section 7.3. We choose F1-score as our detection performance metric due to its simplicity, and its measurement reflects the reality under unbalanced datasets like in our case. Besides F1-score, we could use other metrics like ROC, precision, or recall. The general decision strategy is simple and effective in system development. It provides a firm decision for every sample. On the other hand, the security

practitioner might prefer dealing with decisions that have associated confidence values and filter out less-confident decisions for further investigation. In a real deployment, we would like to have as many as possible detection decisions with high confidence and filter out the few uncertain apps that have low confidence probability. Unfortunately, the general decision strategy does not provide such functionality. For this purpose, we propose the **confidence decision strategy**, a mechanism to automatically filter out apps with uncertain decisions. PetaDroid computes a confidence threshold η that achieves not only a high detection performance (F1-score) but also a negligible error rate (false negative and false positive rates) in the validation dataset. In other words, we add the error rate constraint to the system that computes the detection threshold η from X_{valid}. In the deployment, we make confidence-based decision as follow:

$$D_\eta = \begin{cases} Uncentain & Prob_{Mal} < \eta \ \wedge \ Prob_{Ben} < \eta \\ Malware & Prob_{Mal} >= \eta \ \wedge \ Prob_{Mal} > Prob_{Ben} \\ Benign & Prob_{Ben} >= \eta \ \wedge \ Prob_{Ben} > Prob_{Mal} \end{cases}$$

For example, we could fix the error rate to $< 1\%$ and automatically find η that achieves the highest F1-score in the validation set. Our goal is to maximize certain detection decisions on the deployment, which we called the *detection coverage performance* and minimize alerts for uncertain ones that require further analyses and investigation. In our case, the *detection coverage performance* is the percentage of confidence decisions from X_{test}. In Section 7.3, we conduct experiments where we report *general detection performance* metric in order to compare with existing solutions such as [91, 132, 148]. In addition, we report *confidence detection performance* and *detection coverage performance* metrics which we believe are more suitable for real-world deployment. Furthermore, the *confidence decision strategy* is key in PetaDroid retraining process, aiming toward automatic adaptation as will be explained next.

PetaDroid Adaptation Mechanism

In this section, we describe our mechanism to adapt to Android ecosystem changes overtime automatically. The key idea is to re-train the CNN ensemble on new benign and malware samples at every epoch to learn the latest changes. To enhance the automatic adaptation, we leverage

the confidence analysis to collect an extension dataset that captures the incremental change over time. Initially, we train PetaDroid ensemble using $X_{build} = \{X_{train} + X_{valid}\}$. Afterward, PetaDroid leverages the *confidence detection strategy* to build an extension dataset X_{exten} from test dataset X_{test} with high-confidence detected apps. In a real deployment, X_{test} is a stream of Android apps that needs to be checked for maliciousness by the vetting system. The test dataset $X_{test} = \{X_{Certain}, X_{Uncertain}\}$ is composed of apps having a high-confidence decision ($X_{Certain}$ or X_{exten}) and apps having uncertain decisions $X_{Uncertain}$. In the deployment, PetaDroid accumulates high-confidence apps over time to form X_{exten} dataset. At every time epoch, PetaDroid utilizes the extension dataset X_{exten} to extend the original X_{build} and later updates the CNN ensemble models. In our evaluation, and after updating the CNN ensemble, we report **updated general performance** and **updated confidence-based performance**, which are respectively the general and confidence-based performance of the new trained CNN ensemble on X_{test}. They answer the question: what would be the detection performance on $X_{test} = \{X_{Certain}, X_{Uncertain}\}$ after we build the ensemble on $X_{NewBuild} = \{X_{Certain}, X_{build}\}$? In other words, PetaDroid reviews previous detection decisions using the new CNN ensemble and drives new general and confidence-based performance.

In a deployment environment, PetaDroid is continuously receiving new Android apps, whether benign or malware, represented by X_{test} in our evaluation. PetaDroid employs the extension dataset X_{exten} to overcome pattern changes, whether malicious or benign, automatically. Our approach is based on the assumption that Android apps patterns change incrementally with slow progress. Therefore, starting from a relatively small X_{build} dataset,PetaDroid could learn new patterns from new X_{exten} dataset progressively over time. PetaDroid ensemble update is an automatic operation for every period. During off-line analyses, PetaDroid data extension process could be employed to improve the classification result on a fixed test dataset X_{test} starting from a small X_{build}. Our evaluation (Section 7.3.6) shows the effectiveness of our update strategy.

7.2.4 Malware Clustering

In this section, we detail the family clustering system of PetaDroid. PetaDroid clustering aims at grouping the previously detected malicious apps (Section 7.2.3) into highly similar groups

of malicious apps, which are most likely part of the same malware family. PetaDroid clustering process starts from a list of discretized canonical instruction sequences $P = \langle cS_1, cS_2, \ldots, cS_h \rangle$ of the detected malicious apps. We introduce the *InstNGram2Vec* technique and deep neural network auto-encoder to generate embedding digests for malicious apps. Afterward, we cluster malware digests using the DBScan clustering algorithm to generate malware family groups.

InstNGram2Vec

InstNGram2Vec is a technique that maps concatenated instruction sequences to fixed-size embeddings employing NLP bag of words N-grams [61] and feature hashing [173] techniques.

Common N-Gram Analysis (CNG). The common N-gram analysis (CNG) [61], or simply N-gram, has been extensively used in text analyses and natural language processing in general and related applications such as automatic text classification and authorship attribution [61]. N-gram computes the contiguous sequences of n items from a large sequence. In the context of PetaDroid, we compute canonical instructions N-grams on concatenated sequence cP by counting the instruction sequences of size n. Notice that the N-grams are extracted using a forward moving window (of size n) by one step and incrementing the counter of the found features (instruction sequence in the window) by one. The window size n is a hyper-parameter; we fixed $n = 4$ in all our experimentations. N-gram computation takes place simultaneously with the feature hashing in the form of a pipeline to prevent and limit computation and memory overuse due to the high dimensionality of N-grams.

Feature Hashing. PetaDroid employs Feature Hashing (FH) [173] along with N-grams to vectorize cP. The feature hashing algorithm takes as an input cP N-grams generator and the target length L of the feature vector. The output is a feature vector with components x_i and a fixed size L. In our framework, we fix $L = |V|$, where V is the vocabulary dictionary (Section 7.2.2). We normalize x_i using the euclidean norm (also called L2 norm). Applying *InstNGram2Vec* on a detected malicious app cP produces a fixed size hashing vector hv. Therefore, the result is $HV = \{hv_0, hv_1, \ldots, hv_{DMal}\}$, and hashing vector hv for $DMal$ detected malicious apps.

Deep Neural Auto-Encoder and Digest Generation.

We develop a deep neural auto-encoder through stacked hidden layers of encoding and decoding operations, as shown in Table 7.2. The proposed auto-encoder learns the latent representation of Android apps in an unsupervised way. The unsupervised learning of the auto-encoder is done through the reconstruction (Table 7.2) of the unlabeled hashing vectors $HV = \{hv_0, hv_1, \ldots, hv_{DMal}\}$ of random Android apps. Notice that we do not need any labeling during; for the training of PetaDroid auto-encoder, off-the-self Android apps are sufficient.

#	*Layers*	*Options*
01	Linear	#Output=$\|V\|$, Activation=Tanh
02	Linear	#Output=512 , Activation=Tanh
03	Linear	#Output=256 , Activation=Tanh
04	Linear	#Output=128 , Activation=Tanh
05	Linear	#Output=64 , Activation=Tanh
06	Linear	#Output=64 , Activation=Tanh
07	Linear	#Output=128 , Activation=Tanh
08	Linear	#Output=256 , Activation=Tanh
08	Linear	#Output=512 , Activation=Tanh
10	Linear	#Output=$\|V\|$, Activation=Tanh

Table 7.2: Architecture PetaDroid Deep Neural Auto-Encoder

The training goal is to make the auto-encoder learn to efficiently produce a latent representation (or digest) of an Android app hv that keeps the discriminative patterns of malicious and benign Android apps. Formally, the input to the deep neural auto-encoder [116] network is an unlabeled hash vector $HV = \{hv_0, hv_1, \ldots, hv_{DMal}\}$, denoted $\boldsymbol{x'} \in \mathscr{U}$ on which operates the encoder network $f_{\text{enc}} : \mathbb{R}^{|V|} \rightarrow \mathbb{R}^p$, $p = 64$ as shown in Table 7.2 (parameterized by Θ_{enc}) to produce the latent representation $\boldsymbol{z}_{\boldsymbol{x'},\Theta_{\text{enc}}}$, *i.e.*

$$\boldsymbol{z}_{\boldsymbol{x'},\Theta_{\text{enc}}} = f_{\text{enc}}(\boldsymbol{x'}; \Theta_{\text{enc}}) \tag{23}$$

The produced digest, namely $\boldsymbol{z}_{\boldsymbol{x'},\Theta_{\text{enc}}} \in \mathbb{R}^p$, is used by the decoder network $f_{\text{dec}} : \mathbb{R}^p \rightarrow \mathbb{R}^{|V|}$ to rebuild or reconstruct the InstNGramBag2Vec feature vector. The training lost of the auto-encoder network given the unlabeled hv $\boldsymbol{x'}$ is,

$$\tilde{\boldsymbol{x}}' = f_{\text{dec}}(\boldsymbol{z}; \Theta_{\text{dec}}) \tag{24}$$

$\tilde{\boldsymbol{x}}' \in \mathbb{R}^{d \times w}$ denotes the generated reconstruction.

$$\mathscr{L}_{\text{auto}}(\boldsymbol{x}'; \Theta_{\text{enc}}, \Theta_{\text{dec}}) = \|\boldsymbol{x}' - f_{\text{dec}}(\boldsymbol{z}_{\boldsymbol{x}', \Theta_{\text{enc}}}; \Theta_{\text{dec}})\|^2 \tag{25}$$

In the training phase, the gradient-based optimizer minimizes the objective reconstruction function on the InstNGramBag2Vec feature vectors of unlabeled Android apps.

$$(\Theta^*_{\text{enc}}, \Theta^*_{\text{dec}}) = \arg \min_{\Theta_{\text{enc}}, \Theta_{\text{dec}}} \sum_{i=1}^{N_1+N_2} \mathscr{L}_{\text{auto}}(\boldsymbol{x}'_i; \Theta_{\text{enc}}, \Theta_{\text{dec}}) \tag{26}$$

Notice that PetaDroid auto-encode is trained only once during all the experimentation due to it general usage. To this end, PetaDroid employs a trained encoder f_{dec} to produce digests $\boldsymbol{Z} = \{\boldsymbol{z_0}, \boldsymbol{z_1}, \ldots, \boldsymbol{z_{DMal}}\}$ for the detected malicious apps.

Malware Family Clustering

PetaDroid clusters the detected malware digests $\boldsymbol{Z} = \{\boldsymbol{z_0}, \boldsymbol{z_1}, \ldots, \boldsymbol{z_{DMal}}\}$ into groups of malware with hight similarity and most likely belonging the same family. In PetaDroid clustering, we use an **exclusive** clustering mechanism. This mean that we do not cluster all the detected malicious apps. The clustering algorithm only groups highly similar samples and tags the rest as unclustered. This feature is convenient in real-world deployments since we might not always detect malicious apps from the same family, and we would like to have family groups only if there are groups of the sample malware family. To achieve this feature, we employ the *DBScan* clustering algorithm. *DBScan*, in contrast with clustering algorithms such as *K-means*, produces clusters with high confidence. The most important metrics in PetaDroid clustering is the homogeneity of the produces clusters.

7.2.5 Implementation

We build PetaDroid using `Python` and `Bash` programming languages. We use `dexdump`[2] to disassemble Android app DEX code into Dalvik assembly. The tool `dexdump` is a simple, yet very efficient tool, to parse APK file and produce disassembly in a textual form. We develop python

[2]https://tinyurl.com/y4ze8nyy

and bash scripts to parse Dalvik assembly to produce sequences of canonical instructions. Notice that there is no optimization in the preprocessing; in the efficiency evaluation, we only use a single thread script for a given Android app. We implement PetaDroid neural networks, CNN ensemble and auto-encoders, using PyTorch[3]. For clustering, we employ official `hdbscan`[4] implementation. We evaluate the efficiency of PetaDroid on a commodity hardware server (Intel(R) Xeon(R) CPU E5-2630, 2.6GHz). For training, we use *NVIDIA TitanX* Graphic Processing Unit (GPU).

7.3 Evaluation

In this section, we evaluate PetaDroid framework through a set of experiments and settings involving different datasets. We aim to answer questions such as: *What is the detection performance of PetaDroid on datasets with various sizes (Section 7.3.2)? What is the effect of PetaDroid ensemble and build dataset sizes on the overall performance (Section 7.3.2)? What is the performance of family clustering (Section 7.3.3)? How efficient is PetaDroid in terms of runtime on commodity machines (Section 7.3.7)? How robust is PetaDroid against common obfuscation techniques (Section 7.3.4)?*

7.3.1 Android Dataset

Our evaluation dataset contains 9.7 million Android apps (the dataset size is 100 Tera bytes) collected across the last ten years from August 2010 to August 2019, as depicted in Table 7.3. The extensive coverage in size (9.7 M), time range (06-2010 to 08-2019), and malware families (+300 family) make the result of our evaluation quite compelling. First, we leverage reference Android malware datasets namely: MalGenome [203], Drebin [73], MalDozer [132], and AMD [185]. We use these datasets in the evaluation of both PetaDroid detection and family clustering because they have family labels. A reference dataset helps comparing our evaluation results with the related work. Also, we collected Android malware from VirusShare [5] malware repository. This dataset serves in the evaluation of PetaDroid detection. For benign apps, we randomly samples from the AndroZoo

[3]https://pytorch.org
[4]https://en.wikipedia.org/wiki/DBSCAN
[5]https://VirusShare.com

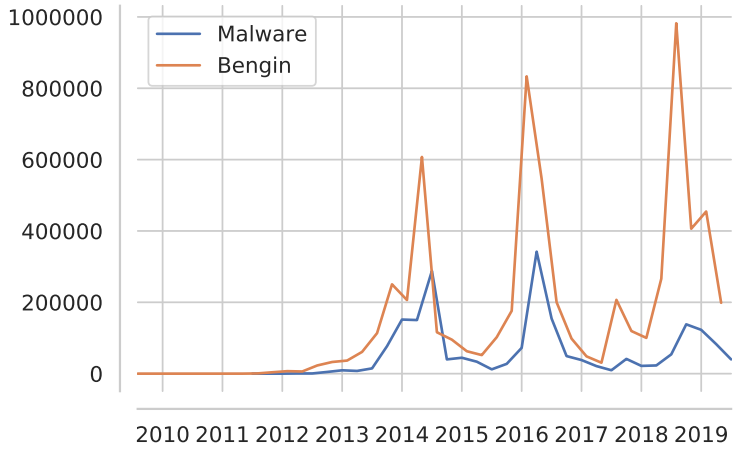


Figure 7.4: AndroZoo Benign and Malware Distribution over Time

[67] dataset (7.4 Million benign samples) two to seven times the size of reference malware dataset in each experiment.

Name	#Samples	#Families	Time
MalGenome [203]	1.3K	49	2010-2011
Drebin [73]	5.5k	179	2010-2012
MalDozer [132]	21k	20	2010-2016
AMD [185]	25k	71	2010-2016
VirusShare [6]	33k	/	2010-2017
MaMaDroid [158]	40k	/	2010-2017
AndroZoo [67]	9.5M	/	2010- Aug 2019

Table 7.3: Evaluation Datasets

In the comparison between PetaDroid, MaMaDroid [148, 158], and DroidAPIMiner [60], we apply PetaDroid on the same dataset (benign and malware) used in MaMaDroid evaluation[7] to measure the performance of PetaDroid against state-of-the-art Android malware detection solutions.

In our use cases, we employ the whole AndroZoo[8] [67] dataset (the collection ends August 2019), which contains 7.4 million benign apps and 2.1 million malware apps. We rely on VirusTotal detection of multiple anti-malware vendors in (metadata provided by AndroZoo repository) to label the samples. As shown in Figure 7.4, the dataset covers more than ten years. To asses PetaDroid

[7] https://bitbucket.org/gianluca_students/mamadroid_code/src/master/
[8] https://androzoo.uni.lu/

obfuscation resiliency, we conduct an obfuscation evaluation on PRAGuard dataset[9], which contains $11k$ obfuscated malicious apps using common obfuscation techniques [147]. In addition, we generate over $100k$ benign and malware obfuscated Android apps employing DroidChameleon obfuscation tool [162] using common obfuscation techniques and their combinations.

7.3.2 Malware Detection

In this section, we report the detection performance of PetaDroid and the effect of hyper-parameters on malware detection performance.

Detection Performance

Table 7.4 shows PetaDroid *general* and *confidence-based* performance in terms of F1-score, recall and precision metrics on the reference datasets. In the general performance, PetaDroid achieves high F1-score $96 - 99\%$ with low false positive rate (recall score of $95.7 - 99.5\%$). The detection performance is higher under confidence settings. The F1-score is 99% and very low false-positive rate with a recall score of 99.8% on average. The confidence-based performance causes the filtration of $1 - 8\%$ low confidence samples from the testing set. In all our experiments, the confidence performance flags $\approx 6\%$ on average, as uncertain decisions, which is a small and realistic value in a deployment with low false positives. The filtered Android apps are flagged as suspicious apps, which might need further attention from the security practitioner.

Name	General (%)	Confidence (%)
	F1 - P - R	F1 - P - R
Genome	99.1 - 99.5 - 98.6	99.5 - 100. - 99.0
Drebin	99.1 - 99.0 - 99.2	99.6 - 99.6 - 99.7
MalDozer	98.6 - 99.0 - 98.2	99.5 - 99.7 - 99.4
AMD	99.5 - 99.5 - 99.5	99.8 - 99.7 - 99.8
VShare	96.1 - 96.4 - 95.7	99.1 - 99.7 - 98.6

Table 7.4: General and Confidence Performances on Various Reference Datasets

Dataset Size Effect

One of the advantages of fragment-based malware detection is the data augmentation of the building dataset by random shuffles. PetaDroid, as shown in Table 7.5, exploits this feature to

[9]http://pralab.diee.unica.it/en/AndroidPRAGuardDataset

enhance the detection performance on small build datasets. In Table 7.5, there is a small change in the detection performance when the build set percentage drops from 90% to 50% from the overall dataset. Note that the build dataset is already composed of 80% training and 20% validation set $X_{build} = \{X_{train}, X_{valid}\}$, which makes the model trained on a smaller dataset. However, PetaDroid detection still perform well under these settings. Notice that in all our experiments, we use 50% from the evaluation dataset as a build dataset.

	General (F1 %)	**Confidence (F1 %)**
Build Dataset Size (%)	50% - 70% - 90%	50% - 70% - 90%
Genome	98.8 - 99.1 - 98.8	100. - 99.5 - 99.1
Drebin	98.2 - 99.1 - 99.1	99.6 - 99.6 - 99.8
MalDozer	98.3 - 98.6 - 98.7	99.6 - 99.5 - 99.6
AMD	99.3 - 99.5 - 99.5	99.7 - 99.8 - 99.7
VShare	95.6 - 96.1 - 96.4	99.0 - 99.1 - 99.1

Table 7.5: Effect of Building Dataset Size on the Detection Performance

Ensemble Size Effect

Another important factor that affect PetaDroid malware detection performance is the number of CNN models in the detection ensemble. Table 7.6 depicts PetaDroid performance under different ensemble sizes. We notice the high detection accuracy using single CNN model ($95 - 99\%$ F1-score). In addition to the strength of CNN in discriminating Android malware, fragment detection adds a significant value the overall performance even in a single CNN mode. In the case of MalGenome (Table 7.6), the ensemble size adds no value to the detection performance due to small size of MalGenome dataset ($1.3k$ malware + $12k$ benign randomly sampled from AndroZoo [67]). In case of VirusShare (Table 7.6), augmenting the ensemble size enhanced the detection rate. Our empirical tests show that $\phi = 6$ as the ensemble size gives good detection results. Increasing beyond $\phi = 6$ will have negligible benefits.

	General (F1 %)	**Confidence (F1 %)**
#Model	1 - 5 - 10 - 20	1 - 5 - 10 - 20
MalGenome	99.5 - 99.3 - 99.3 - 99.1	99.5 - 99.5 - 99.5 - 99.5
Drebin	99.0 - 99.1 - 99.0 - 99.1	99.4 - 99.6 - 99.6 - 99.6
MalDozer	98.0 - 98.6 - 98.4 - 98.6	99.0 - 99.5 - 99.5 - 99.5
AMD	99.3 - 99.5 - 99.5 - 99.5	99.5 - 99.8 - 99.7 - 99.8
VShare	95.0 - 96.0 - 96.1 - 96.1	98.1 - 99.0 - 99.2 - 99.1

Table 7.6: Effect of Ensemble Size on Detection

7.3.3 Family Clustering

In this section, we present the results of PetaDroid family clustering on reference datasets. Malware family clustering phase comes after PetaDroid detects a considerable number of malicious Android apps. The number of detected apps could vary from $1k$ (MalGenome [203]) to $24k$ (AMD [185]) samples depending on the deployment. We use *homogeneity* [165] and *coverage* metrics to measure the family clustering performance. The homogeneity metric scores the purity of the produced family clusters. A perfect homogeneity means each produced cluster contains samples from only one malware family. PetaDroid clustering aims only to generate groups with confidence-based while ignoring less certain groups. The coverage metrics score the percentage of the clustered dataset with confidence.

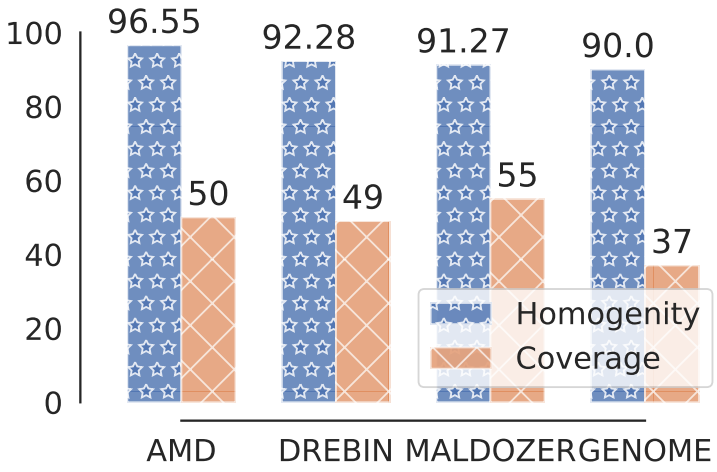


Figure 7.5: Clustering Performance on Reference Datasets

Figure 7.5 summarize the clustering performance in terms of *homogeneity* and *coverage* scores. PetaDroid can produce clusters with high *homogeneity* $90 - 96\%$ while keeping an acceptable *coverage*, 50% on average. At first glance, 50% *coverage* seems to be a modest result, but we argue that it is satisfactory because: (i) we could extend the coverage, but this might affect the quality of the produced clusters. In the deployment, high confidence clusters with minimum errors and acceptable coverage might be better than perfect coverage (in case of K-means clustering algorithm) with a high error rate. (ii) The evaluation datasets have long tail malware families, meaning that most families have only a few samples. This makes the clustering very difficult due to the few samples

(less than five samples) in each malware family in the detected dataset. In a real deployment, we could add unclustered samples to the next clustering iterations. In this case, we might accumulate enough samples to cluster for the long tail malware families.

7.3.4 Obfuscation Resiliency

In this section, we report PetaDroid detection performance on obfuscated Android apps. We experiment on: (1) PRAGuard obfuscation dataset [147] ($10k$) and (2) obfuscation dataset generated using DroidChameleon [162] obfuscation tool ($100k$). In the PRAGuard experiment, we combine Praguard dataset with $20k$ benign Android apps randomly sampled from benign app of AndroZoo repository. We split the dataset equally into build dataset $X_{build} = \{X_{train}, X_{valid}\}$ and test dataset X_{test}. Table 7.7 presents the detection performance of PetaDroid on different obfuscation techniques. PetaDroid shows high resiliency to common obfuscation techniques by having almost perfect detection rate, 99.5% F1-score on average.

		General Performance (%)		
ID	**Obfuscation Techniques**	**F1 (%)**	**P (%)**	**R (%)**
1	`Trivial`	99.4	99.4	99.4
2	`String Encryption`	99.4	99.3	99.4
3	`Reflection`	99.5	99.5	99.5
4	`Class Encryption`	99.4	99.4	99.5
5	`(1) + (2)`	99.4	99.4	99.4
6	`(1) + (2) + (3)`	99.4	99.3	99.5
7	`(1) + (2) + (3) + (4)`	99.5	99.4	99.6
	Overall	99.5	99.6	99.4

Table 7.7: PetaDroid Obfuscation Resiliency on PRAGuard Dataset

In the DroidChameleaon experiment, we evaluate PetaDroid on other obfuscation techniques, as shown in Table 7.8. The generated dataset contains obfuscated benign (apps originally from AndroZoo) and malware samples (originally from Drebin). In the building process of CNN ensemble, we only train with one obfuscation technique (Table 7.8) and make the evaluation on the rest of the obfuscation techniques. Table 7.8 reports the result of obfuscation resiliency on DroidChameleon generated dataset. The results show the robustness of PetaDroid. According to this experiment, PetaDroid is able to detect malware obfuscated with common techniques even if the training is done on non-obfuscated datasets. We believe that PetaDroid obfuscation resiliency comes from the usage of (1) Android API (canonical instructions) sequences as features in the machine learning

development. Android APIs are crucial in any Android app. A malware Developer cannot hide API access, for example, *SendSMS* unless the malicious payload is downloaded at runtime. Therefore, PetaDroid is resilient to common obfuscations as long as they do not remove or hide API access calls. (2) The other factor is fragment-randomization, which makes PetaDroid models robust to code transformation and obfuscation in general. We argue that training machine learning models on dynamic fragments enhances the resiliency of the models against code transformation.

	General Performance		
Obfuscation Techniques	**F1 (%)**	**P (%)**	**R (%)**
No Obfuscation	99.7	99.8	99.6
`Class Renaming`	99.6	99.6	99.5
`Method Renaming`	99.7	99.7	99.7
`Field Renaming`	99.7	99.8	99.7
`String Encryption`	99.8	99.8	99.7
`Array Encryption`	99.8	99.8	99.7
`Call Indirection`	99.8	99.8	99.7
`Code Reordering`	99.8	99.8	99.7
`Junk Code Insertion`	99.8	99.8	99.7
`Instruction Insertion`	99.7	99.8	99.7
`Debug Information Removing`	99.8	99.8	99.7
`Disassembling and Reassembling`	99.8	99.8	99.7

Table 7.8: PetaDroid Obfuscation Resiliency on DroidChameleon Generated Dataset

7.3.5 Change over Time Resiliency

An important feature in modern Android malware detection is the resiliency to change over time [132, 148, 158]. We study the resiliency of PetaDroid over the last seven-year (2013-2019). We randomly sample from AndroZoo repository a number of $10k$ Android apps ($5k$ malware and $5k$ benign apps) for each year (2013-2019). As result, we have $70k = 35k_{Mal} + 35k_{Ben}$. We build the CNN ensemble using year Y_x samples and evaluate on the other years $Y_{1..N}$ samples. Figure 7.6(a) shows the general and the confidence performances of PetaDroid, for models trained on 2013 samples, in terms of F1-score on 2014-2019 samples. As shown in Figure 7.6(a), PetaDroid , trained on 2013 dataset, achieved 98.17%, 96.10%, 93.01%, 70.60%, 54.82%, 55.59% F1-score on 2014, 2015, 2016, 2017, 2018, and 2019 datasets respectively. PetaDroid sustains a relatively good performance over the first few years. In 2018 and 2019, the performance drops considerably. In comparison to MaMaDroid [158], PetaDroid shows a higher time resiliency over seven years, while MaMaDroid drops considerably in year three (40% F1-score on year four). Figure 7.6(b)

shows the training is on 2014 samples, which shows a performance enhancement over the overall evaluation period. The overall performance tends to increase as we train on a recent year dataset as depicted in Figure 7.6(c), 7.6(d), and 7.6(e). In Figure 7.6(f) and 7.6(g), training is on samples from 2018 and 2019 respectively, PetaDroid performance slightly decreases on old samples form 2013 and 2014. Our interpretation is that old and deprecated Android APIs are not present in new apps from 2018 and 2019, which we use for the training and this influences negatively the detection performance.

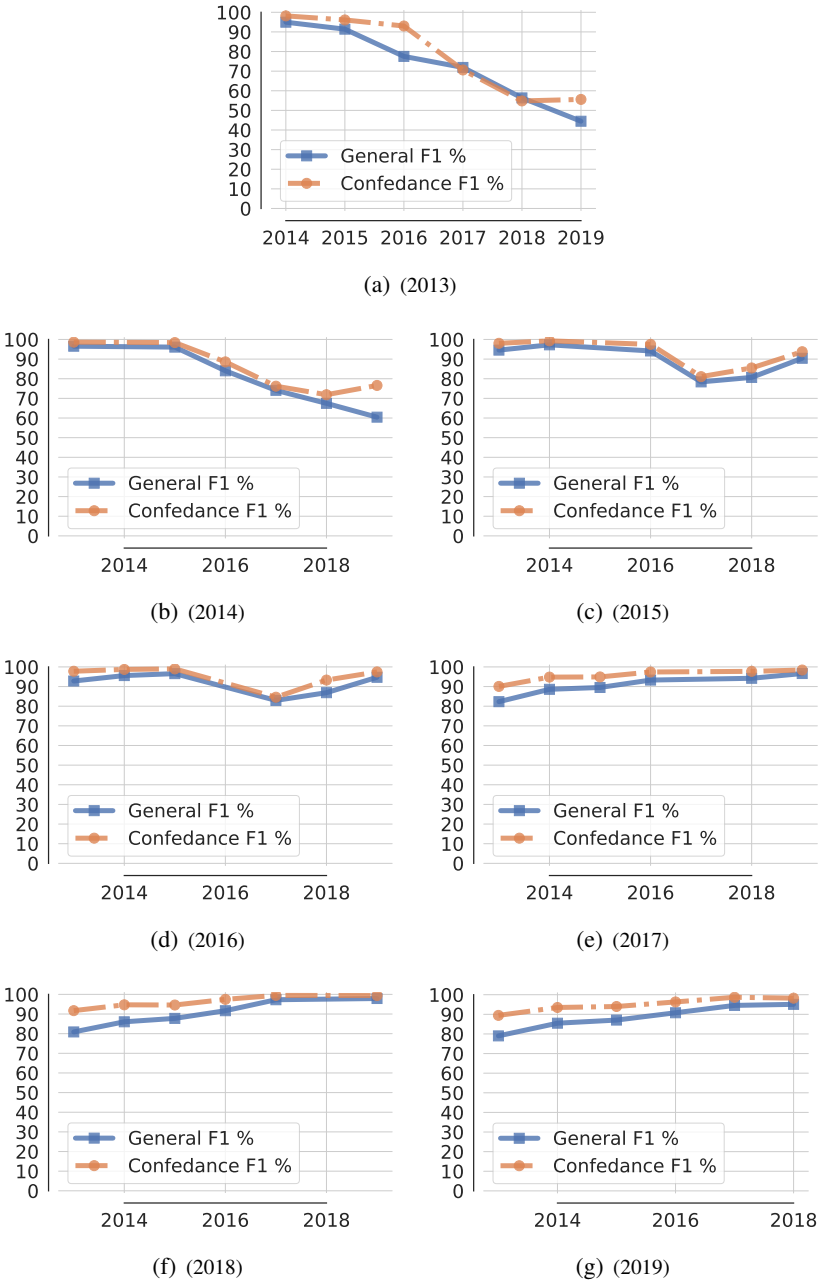
We take from this experiment that PetaDroid is resilient to change over time for years $t \pm 2$ when we train on year Y_t samples. PetaDroid covers about five years $\{Y_{t-2}, Y_{t-1}, Y_t, Y_{t+1}, Y_{t+2}\}$ of Android app change.

7.3.6 PetaDroid Automatic Adaptation

PetaDroid automatic adaptation goes beyond time resiliency. PetaDroid employs the confidence performance to collect an extension dataset X_{extend} during the deployment. PetaDroid automatically uses X_{extend} in addition to the previous build dataset as a new build dataset $X_{build(t)} = X_{build(t-1)} \cup X_{extend}$ to build a new ensemble at every new epoch. Table 7.9 depicts PetaDroid performance with and without automatic adaptation. PetaDroid achieves very good results compared to the previous section (Figure 7.6(a)). PetaDroid maintains an F1-score in the range of $83 - 95\%$ during all years. Without adaption, PetaDroid F1-score drops considerably starting from 2017 samples. Table 7.9 shows the performance of revisiting detection decisions on previous Android apps X_{test} (benign and malware) after updating PetaDroid ensemble using $X_{build} \cup X_{extend}, X_{extend} \subseteq X_{test}$. The update performance is significantly enhanced in the overall detection during all years. Revisiting malware detection decisions is common practice in App market (periodic full or partial scan the market's apps), which empowers the use case of PetaDroid automatic adaptation feature and the update metric.

7.3.7 Efficiency

In Figure 7.7, we depict the average time of PetaDroid detection process. The latter include disassembly, preprocessing, and inference time. PetaDroid spends, on average, 4.0 seconds to



(a) (2013)

(b) (2014)

(c) (2015)

(d) (2016)

(e) (2017)

(f) (2018)

(g) (2019)

Figure 7.6: PetaDroid Resiliency to Changes over Time

fingerprint an Android app. The runtime increases for benign apps, 5.5 seconds, because their package sizes tend to be larger compared to malicious ones. For malware apps, PetaDroid spends, on average, 3.0 seconds for fingerprinting on the app. The detection process of benign apps takes more time on the average compared with malicious apps because benign apps tends to have larger

Year	No Update(F1%)	General(F1%)	Confidence(F1%)	Update(F1%)
2014	98.2	97.0	97.9	99.7
2015	96.1	95.8	96.7	97.5
2016	93.0	93.3	94.8	96.4
2017	70.6	83.9	84.2	95.4
2018	54.8	87.6	91.6	93.8
2019	55.6	96.3	98.7	99.1

Table 7.9: Performance of PetaDroid Automatic Adaptation

size than malicious apps.

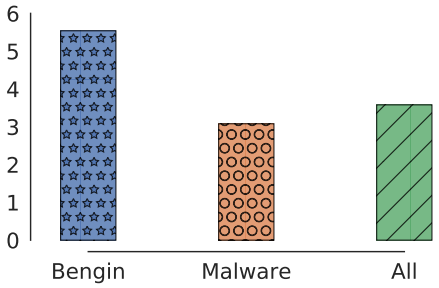


Figure 7.7: PetaDroid Runtime Efficiency

7.4 Comparative Study

In this section, we conduct a comparative study between PetaDroid and state-of-the-art Android malware detection systems namely, MaMaDroid [148, 158] and DroidAPIMiner [60]. Our comparison is based on applying PetaDroid on the same dataset (malicious and benign apps) and settings that MaMaDroid used in the evaluation (provided by the authors in [158]). The dataset is composed of 8.5K benign and 35.5K malicious apps in addition to Drebin [73] dataset. The malicious samples are tagged, by time; malicious apps from 2012 (Drebin), 2013, 2014, 2015, and 2016 and benign apps are tagged as oldbenign and newbenign, according to MaMaDroid evaluation.

7.4.1 Detection Performance Comparison

Table 7.10 depicts the direct comparison between MaMaDroid and PetaDroid different dataset combinations. In PetaDroid, we present the general and the confidence performance in terms of F1-score. For MaMaDroid and DroidAPIMiner, we present the original evaluation result [158] in terms of F1 score, which are equivalent to the general performance in our case. Notice that, we

present only the best results of MaMaDroid and DroidAPIMiner as reported in [158].

	Peta (F1%) General-Confidence	MaMa (F1%)	Miner (F1%)
drebin&oldbenign	**98.94 - 99.40**	96.00	32.00
2013&oldbenign	**99.43 - 99.81**	97.00	36.00
2014&oldbenign	**98.94 - 99.47**	95.00	62.00
2014&newbenign	**99.54 - 99.83**	99.00	92.00
2015&newbenign	**97.98 - 98.95**	95.00	77.00
2016&newbenign	**97.44 - 98.60**	92.00	36.00

Table 7.10: Detection Performance of MaMaDroid, PetaDroid, and DroidAPIMiner

As depicted in Table 7.10, PetaDroid outperforms MaMaDroid and DroidAPIMiner in all datasets in the general performance. The detection performance gap increases with the confidence-based performance. Notice that the coverage in the confidence-based settings is almost perfect (only few apps have been filtered due to the low confidence) for all the experiments in Table 7.10.

7.4.2 Efficiency Comparison

In Table 7.11, we report the required average time for MaMaDroid and PetaDroid to fingerprint one Android app. PetaDroid takes 03.58 ± 04.21 seconds on average for the whole process (DEX disassembly, assembly preprocessing, CNN ensemble inference). MaMaDroid, compared to PetaDroid, tends to be slower due to the heavy preprocessing. MaMaDroid preprocessing [148] is composed of the call graph extraction, sequence extraction, and Markov change modeling, which require 25.40 ± 63.00, 1.73 ± 3.2, 6.7 ± 3.8 seconds respectively for benign samples and 09.20 ± 14.00, 1.67 ± 3.1, 2.5 ± 3.2 seconds respectively for malicious samples. On average, PetaDroid ($3.58s$) is approximately eight times faster than MaMaDroid (≈ 23).

	PetaDroid (seconds)	**MaMaDroid** (seconds)
Malware	02.64 ± 03.94	$09.20 \pm 14.00 + 1.67 \pm 3.1 + 2.5 \pm 3.2$
Bengin	05.54 ± 05.12	$25.40 \pm 63.00 + 1.73 \pm 3.2 + 6.7 \pm 3.8$
Average	03.58 ± 04.21	$\approx 23s$

Table 7.11: MaMaDroid and PetaDroid Runtime

7.4.3 Time Resiliency Comparison

MaMaDroid evaluation emphasizes the importance of time resiliency for modern Android malware detection. Table 7.12 depicts the performance with different dataset settings, such as training

using an old malware dataset and testing on a newer one. PetaDroid outperforms (or obtains a very similar result in few cases) MaMaDroid and DroidAPIMiner in all settings. Furthermore, the results show that PetaDroid is more robust to time resiliency compared to MaMaDroid [158].

Testing Sets	drebin & oldbenign			2013 & oldbenign			2014 & oldbenign			2015 & oldbenign			2016 & oldbenign		
Training Sets	**Miner**	**MaMa**	**Peta**	**Miner**	**MaMa**	**Peta**	**Miner**	**MaMa**	**Peta**	**Miner**	**MaMa**	**Peta**	**Miner**	**MaMa**	**Peta**
drebin&oldbenign	32.0	96.0	**99.4**	35.0	95.0	**98.6**	34.0	72.0	**77.5**	30.0	39.0	**44.0**	33.0	42.0	**47.0**
2013&oldbenign	33.0	94.0	**97.8**	36.0	97.0	**99.6**	35.0	73.0	**85.4**	31.0	37.0	**59.3**	33.0	28.0	**56.6**
2014&oldbenign	36.0	92.0	**95.8**	39.0	93.0	**98.6**	62.0	95.0	**99.4**	33.0	78.0	**91.4**	37.0	75.0	**88.9**
	drebin & newbenign			2013 & newbenign			2014 & newbenign			2015 & newbenign			2016 & newbenign		
Training Sets	**Miner**	**MaMa**	**Peta**	**Miner**	**MaMa**	**Peta**	**Miner**	**MaMa**	**Peta**	**Miner**	**MaMa**	**Peta**	**Miner**	**MaMa**	**Peta**
2014&newbenign	76.0	98.0	**99.3**	75.0	98.0	**99.7**	92.0	99.0	**99.8**	67.0	85.0	**91.4**	65.0	81.0	**82.1**
2015&newbenign	68.0	97.0	**97.1**	68.0	97.0	**97.8**	69.0	**99.0**	98.9	77.0	95.0	**99.0**	65.0	88.0	**95.4**
2016&newbenign	33.0	**96.0**	95.6	35.0	98.0	**98.2**	36.0	**98.0**	97.9	34.0	92.0	**95.2**	36.0	92.0	**98.3**

Table 7.12: Classification performance of MaMaDroid, PetaDroid, DroidAPIMiner.

7.5 Case Studies

In this section, we conduct mega-scale experiments on AndroZoo dataset (9.5 million Android apps). We argue that these experiments reflect real word deployments due to the dataset size, time distribution (2010-2019), and malware family diversity. We report that PetaDroid detection overall performance and overtime performance using our automatic adaptation feature in terms of general confidence.

7.5.1 Scalable Detection

In this experiment, we employ 8.5 out of 9.5 million Android apps from AndroZoo dataset. The used dataset is composed of 1.0 million malicious samples and 7.5 millions benign sample. We filter out app samples that do not correlate with VirusTotal, or they have less than five maliciousness flags in VirusTotal. In our experiments, we randomly sample k samples as build dataset X_{build} and use the rest $8.5M - k$ as X_{test}. We use different k sizes, $k \in \{10k, 20k, 50k, 70k, 100k\}$, and we repeat each experiment ten times to compute the average detection performance. In Table 7.13, we report the detection performance in terms of F1-score of PetaDroid on AndroZoo dataset. PetaDroid shows a high F1-score for all the experiments, $95 - 97\%$ F1-score. We achieved 95.34% F1-score when the build set is only $10k$. We argue that fragment randomization plays an important role in achieving these detection results because it acts as a data augmenter (the randomization generates

several canonical instruction sequences from a given Android app through the permutation of the methods code) during the training phase.

#Samples	General (F1 %)	Confidence (F1 %)
10k	95.34	97.88
20k	96.17	98.01
50k	96.50	98.10
70k	96.76	98.11
100k	97.04	98.17

Table 7.13: PetaDroid Mega-Scale Detection Performance

7.5.2 Scalable Automatic Adaptation

In this experiment, we put the automatic adaptation feature on mega scale test using 5.5 million samples from AndroZoo dataset (2013-2016) on 25 training epochs (every three months). We initiate PetaDroid on only $25k$ build dataset collected between $2013-Jan-01$ and $2013-Jan-31$. PetaDroid rebuilds new CNN ensemble for each three month samples by retraining on $X_{build(t)} = X_{build(t-1)} \cap X_{extend}$.

	Before Update (F1 %)		After Update (F1 %)	
Update Epoch	General	Confidence	General	Confidence
2013-01-31	/	/	/	/
2013-04-30	96.02	98.43	99.01	99.71
2013-07-31	94.52	96.12	97.95	99.56
2013-10-31	94.42	97.37	97.03	99.56
2014-01-31	83.45	92.74	95.45	99.37
2014-04-30	90.48	96.21	94.15	99.43
2014-07-31	86.98	95.79	91.53	99.11
2014-10-31	92.32	98.47	93.11	99.00
2015-01-31	91.57	97.72	90.91	99.18
2015-04-30	91.31	98.55	92.72	99.09
2015-07-31	88.16	97.46	88.90	98.99
2015-10-31	73.82	87.45	83.44	97.57
2016-01-31	78.59	90.92	85.11	96.26
2016-04-30	84.78	95.44	86.38	98.33
2016-07-31	71.39	88.08	80.27	93.54
2016-10-31	78.31	85.79	79.68	90.75

Table 7.14: Autonomous Adaptation on Mega-Scale Dataset

In Table 7.14, we report the general and confidence performance before and after updating PetaDroid CNN ensemble on an extended build dataset. The automatic adaption feature achieves very good results. The general and confidence-based performance in terms of F1 score vary between $71.39-96.02\%$ and $85.79-98.55\%$, respectively. These performance results increase considerably ($90.75 - 99.71\%$ F1-score) after revising the previous detection decisions using an updated CNN

ensemble using a new X_{extend} on each epoch.

7.6 Summary

In this chapter, we presented PetaDroid, an Android malware detection, and family clustering framework for large scale deployments. PetaDroid employs supervised machine learning, an ensemble of convolutional neural networks on top of *Inst2Vec* features, to fingerprint Android malicious apps accurately. Furthermore, PetaDroid uses unsupervised machine learning, precisely DBScan clustering on top of *InstNGram2Vec* and deep auto-encoders features, to cluster highly similar malicious apps into their most likely malware family groups. In PetaDroid, we introduced fragment-based detection, in which we randomize the macro-action of Android APIs while keeping the inner order of methods' sequences. Fragment randomization acts as a data augmentation mechanism during the training and strengthens detection robustness against common obfuscation techniques during deployment. Also, we introduced the automatic adaption technique that leverages confidence-based decision making to build a new CNN ensemble on confidence detection samples. The adaptation technique automatically enhances PetaDroid time resiliency. We conducted a thorough evaluation of different reference datasets and various settings. PetaDroid achieved high detection (98-99% F1-score) and family clustering (96% cluster homogeneity) performance. Our comparative study between PetaDroid and MaMaDroid [148, 158] shows that PetaDroid outperforms state-of-the-art solutions on various settings. We evaluate PetaDroid on a market scale Android dataset, over $100TB$ of data and 9.7 million samples.

In the next chapter, we apply the learned techniques from Android malware fingerprinting and detection on an entirely different platform and malware type, namely ransomware. The goal is to check the applicability of the elaborated techniques on general malware.

Chapter 8

Ransomware Hybrid Fingerprinting

8.1 Overview

In this chapter, we present SwiftR, a novel ransomware fingerprinting framework for ransomware. SwiftR employs the elaborated methods, techniques, and tools in the previous chapters to detect ransomware. Ransomware attacks have recently seen a dramatic increase. In February 2015, TeslaCrypt [37] ransomware targeted online gaming and collected over 75k$. In May 2015, Fusob [57], one of the major mobile ransomware, gathered ransoms from half of the infected devices. Tox [30] is the first ransomware-as-a-service (RaaS) kit that was published in May 2015. Later, in September 2015, Chimera [50] was identified as the first ransomware that threatens system owners by leaking encrypted data. In February 2016, Locky [14] attacked hospitals in Hollywood, collecting 17k$. Then, in April 2016, the first version of Petya [49], delivered via Dropbox cloud storage, was discovered. A more advanced version of Petya [31] has been identified in June 2017, which exploits a modified EternalBlue exploit [43] for spreading over the Internet. In May 2017, WannaCry [32] infected over 300 thousand systems in over 150 countries. In October 2017, Bad Rabbit [16] targeted Ukraine's Ministry of Infrastructure and Kiev's public transportation system. In March 2018 [42], the city of Atlanta was crippled for six days after being hit by a ransomware that targeted the IT services of the city. Also, since its appearance in August 2018, Ryuk [59] ransomware, has attacked many organizations around the world.

We identify the following problems:

P1: Most of the available ransomware detection solutions employ dynamic analysis as a source of fingerprinting features. Dynamic analysis [90, 134] provides an effective way to fingerprint ransomware, but it suffers from the fact that most of the execution paths of the analyzed sample are not covered by anti-sandboxing techniques. Furthermore, it is a time-consuming task, especially given the large number of ransomware and malware that are collected on a daily basis [8, 9]. However, these gaps could be narrowed by leveraging a hybrid analysis, whereby both static and dynamic analysis are employed. Although static analysis is known for not being resilient against code obfuscations, hybrid analysis can inherit the strengths of both static and dynamic analysis, which increases resiliency against detection evasion, and hence improves the effectiveness and the efficiency of ransomware detection.

P2: When employing conventional machine learning, human intervention is required in terms of feature engineering to manually select the relevant features for malware detection. However, this issue can be addressed by adopting a deep learning approach, which avoids the process of hand-crafted feature engineering by learning a set of features automatically. This will help in improving the automation process of building and deploying malware detection systems.

P3: The state-of-the-art malware detection (including ransomware detection) solutions are limited to a specific platform. However, the diversity of architectures and platforms highlights the need for cross-platform and heterogeneous architecture malware/ransomware detection. This portability is a requirement in case of static analysis due to the presence of different architectures. Also, given the variety of platforms, dynamic analysis produces different types of behavioral analysis reports. This also requires portable detection in order to handle the variety in report types.

P4: Most of the existing solutions focus on ransomware detection. They extract security features to distinguish between ransomware and benign code. However, less importance is given to other detection tasks: (1) segregation between ransomware and other malware (that we will call attribution), and ransomware family attribution. *Malware attribution* is the task of differentiating ransomware from other malware. Family attribution assigns the detected ransomware to known ransomware family. These different aspects provide granular intelligence for better security-oriented decision making.

Motivated by the aforementioned problems, we propose in this chapter, SwiftR, which represents a novel framework for ransomware, malware, and family fingerprinting using raw hybrid features. The proposed framework allows to detect ransomware, distinguish them from general malware, and infer their ransomware family with high accuracy across different architectures, compilers, and operating platforms. SwiftR is composed of two novel maliciousness fingerprinting stages: static SwiftR, and dynamic SwiftR. Static SwiftR is the first stage of SwiftR framework that leverages the static analysis features extracted from the Intermediate Representation (IR). Dynamic SwiftR is the second stage of SwiftR framework. Dynamic SwiftR is triggered when the Static SwiftR stage provides a low probability confidence. We execute the sample in a sandboxing environment (platform-dependent) and collect the behavioral analysis reports.

8.1.1 Threat Model

We position SwiftR as a ransomware detection system that relies on both static and dynamic analyses. Therefore, SwiftR inherits the strengths of both analyses. Static SwiftR provides high code coverage and exploration of the execution paths. More importantly, Static SwiftR provides swift ransomware detection due to the efficiency of the stages (feature extraction, feature representation, neural network detection) in the detection pipeline. Dynamic SwiftR is the second stage in SwiftR framework, and enhances the resiliency of the system to sophisticated binary packing [123, 149] and code obfuscations. We employ flat implicit features such as IR VEX operation embedding sequence that provides resiliency to code transformation, as presented in Asm2vec [100]. Furthermore, relying on a neural network helps in discovering hidden and powerful features to identify ransomware that is more resilient to obfuscation [100], compared to manual engineered features. Despite the previous obfuscation mitigation measures, Static SwiftR is not immune against sophisticated packing and obfuscation techniques. In this regard, Dynamic SwiftR allows to bridge such a gap in the case where Static SwiftR generates low confidence results.

8.1.2 Static SwiftR

Crypto-ransomware has distinguishable behaviors compared to benign binaries and general malware. This is due to the heavy usage of cryptographic primitives to encrypt the victim's files. Furthermore, specific ransomware families leverage known vulnerabilities to spread to other machines in the network. For example, WannaCry family [43] exploits the EternalBlue vulnerability to infect other local machines. These symptoms have particular fingerprints in the ransomware binary and its assembly. For example, WannaCry samples use a combination of RSA and AES encryption algorithms. Windows Crypto API is used for RSA encryption and random key generation [15]. Besides, a third-party implementation of AES is statically linked within some variants of WannaCry family [15].

8.1.3 Dynamic SwiftR

The execution of a binary sample (or app) produces textual reports, whether in a controlled environment (software sandbox) or a real environment. The reports, which are sequences of statements are the result of the app events, and this depends on the granularity provided by the execution environment. Furthermore, each report statement represents a sequence of words that gives a more granular description of the actual app event. From a security perspective, malware behaviors are summarized in the execution report, which is a sequence of statements, and each statement is a sequence of words. We argue that ransomware has distinguishable behaviors from general malware and benign apps, and this characteristic is translated into words in the behavioral report.

8.1.4 SwiftR Neural Network

Ransomware mitigation solutions based on dynamic analysis such as those in [93, 134] are based on manual feature engineering. These features depend on a specific platform and cannot be generalized to other platforms and architectures. Also, such an approach is not keeping up with the pace of change in ransomware and general malware. In contrast, we use raw representations of labeled samples and machine learning techniques in order to extract and filter relevant security features during the training phase. In this chapter, we employ deep learning techniques for the

ransomware fingerprinting. This enables automatic feature engineering through the use of the multi-layer Neural Network (NN) in Static (Section 8.4.2) and Dynamic (Section 8.4.3) SwiftR. Our deep learning approach achieves the highest detection rates without the need for manual security practitioner intervention for feature engineering. We only need to represent the ransomware and non-ransomware samples in their raw sequences format. The neural network will do the automatic feature engineering during the training phase. In Static SwiftR, neural networks with embedding sequences show high resiliency to obfuscations compared to manual feature engineering techniques [100]. In Dynamic SwiftR, we feed dynamic analysis reports into an LSTM neural network in order to perform cross-sandboxing ransomware fingerprinting.

8.2 Methodology

SwiftR methodology involves the composition of Static and Dynamic SwiftR.

8.2.1 Static SwiftR Approach

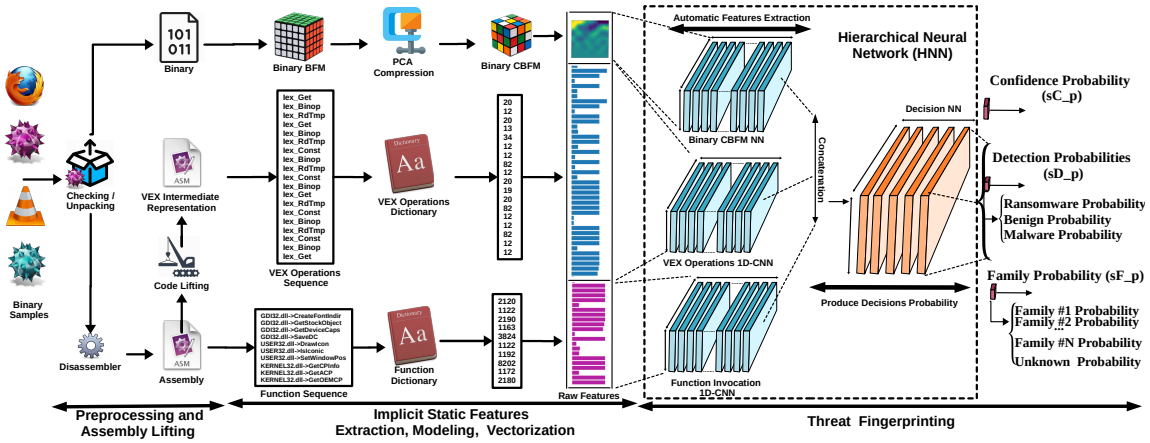


Figure 8.1: Static SwiftR Overview

Static SwiftR, as shown in Figure 8.1, tackles the static side of binary samples such as the assembly code. It provides a portable ransomware detection capability by employing portable and raw features extracted from the assembly code intermediate representation and the sample binary content. SwiftR Static detection process starts by taking a packed binary and unpacking it using

unpacking tools (provided by a third-party security company). Static SwiftR leverages the binary and the underlying assembly static contents to fingerprint ransomware.

Static Implicit Features

As illustrated in Figure 8.1, SwiftR extracts VEX IR operations embedding sequences and the function invocation embedding sequences from the assembly content. We produce the embeddings, by employing state-of-the-art word embedding techniques based on word2vec [153]. The latter considers co-occurrences in the sequences. The intuition is that items that frequently co-occur will have "embeddings" with smaller distance between them, and vice versa. Previous works [100], [132], [193] show the effectiveness of using word embedding techniques, such as word2vec, to fingerprint obfuscated apps in general and specifically malware in particular. From the sample binary content, SwiftR produces the entropy matrix of the whole binary. Previous works [146], [156], [62] show the resiliency of the binary entropy to various obfuscation techniques. Implicit static features will be described in more detail in the next sections.

Static Neural Network Architecture

The previous static features are raw and implicit. We aim to automatically extract explicit and latent features from ransomware samples using deep learning techniques. For this purpose, we design a hierarchical neural network based on different neural network architectures such as convolution neural network (CNN) [113] and Multi-Layer Perceptron (MLP) [113]. Each raw feature type is an input for one neural network block (as depicted in Figure 8.1), which acts as an automatic feature engineering component. All these neural networks are connected to an MLP that serves as a decision maker. We call our custom architecture a Hierarchical Neural Network (HNN). This terminology is our proper name for the proposed NN architecture and it is not related to other neural network architectures with similar names in the literature. The output of HNN represents probability vectors. As shown in Figure 8.1, each component in sD_p captures the likelihood class for a sample, i.e., ransomware, other general malware, or benign binary. Also, HNN outputs ransomware family probability sF_p distribution of known ransomware families. Last, HNN outputs the confidence probability sC_p that measures the confidence in the decision made by the static system.

Static Training and Decision Making

Static SwiftR is a supervised multi-task neural network. We train an HNN on a training dataset to maximize the probability of correctly labeling the samples. Each sample in the training dataset has three label categories: First, we have the detection label that can be either ransomware, other malware, or benign binary executable. Second, we have the ransomware family label that is ascribed to the sample (one or more ransomware families depending on its characteristics). Third, we have another label that we call the confidence label, which indicates the capability of static SwiftR to correctly identify the class of the binary sample. The intention is to train an HNN on highly-obfuscated, encrypted, packed samples (ransomware, other malware, and benign) to output a low-confidence probability sC_p. Otherwise, the HNN will output a high sC_p, which indicates that SwiftR achieves classification with high confidence. As such, Static SwiftR provides measurable insight on the capability of static analysis to accurately fingerprint the class of given binary samples. During deployment, we first ensure that $sC_p > sC_{th}$ (where sC_{th} is the confidence threshold) and if so, we use the resulting sD_p and sF_p as detection and family attribution results. Instead, if the confidence threshold is not met, we resort to Dynamic SwiftR for further analysis.

8.2.2 Dynamic SwiftR Approach

Dynamic SwiftR, as shown in Figure 8.2, focuses on the behavioral reports produced during samples execution in a controlled environment (sandboxing) or in a runtime environment (user machine). It is important to mention that Dynamic SwiftR is portable system as it relies on automatic engineering of relevant security features without the intervention of a security expert. Actually, Dynamic SwiftR is built on top of Natural Language Processing (NLP) modeling and recurrent neural network techniques - specifically LSTM [117]. The key idea is to model a behavioral report, in a manner that is agnostic to the execution environment, as a Sequence of Words (SoW), where the features are the words of the reports. As such, we employ LSTM based neural network to automatically discover relevant security features from a report's sequence of words that can fingerprint ransomware.

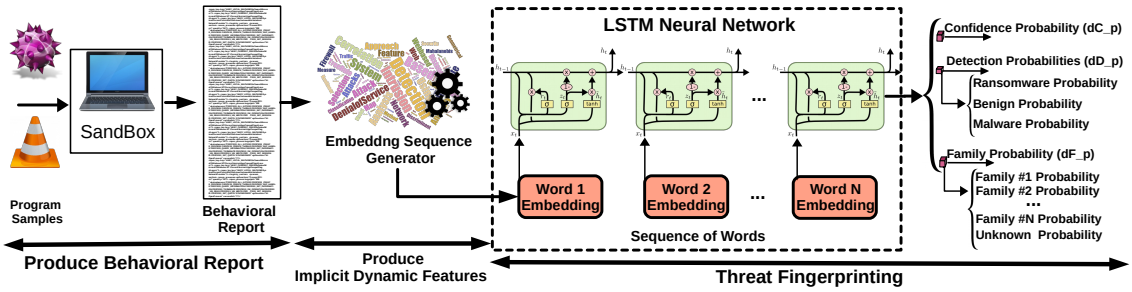


Figure 8.2: Dynamic SwiftR Overview

Producing Behavioral Reports

Dynamic SwiftR Framework starts from a behavioral report, which is serialized from the execution in a controlled environment, as the sample is running. We consider two primary sources for such reports based on the environment type. First, we collect reports, in a controlled system, from a software sandbox environment [189], where we execute a binary program, a malware or a benign executable. Second, we could collect as well behavioral reports from a production system in the form of runtime logs of running programs.

Dynamic Implicit Features

Dynamic SwiftR uses word-centric features to fingerprint ransomware. A behavioral report is converted to a sequence of words with minimal preprocessing, such as removing special characters and splitting sentences. Afterward, we map the sequence of words to the sequence of embeddings using state-of-the-art word embedding techniques such as word2vec [153] as presented in Section 8.2.1.

Dynamic Neural Network Architecture

Dynamic SwiftR uses an RNN to achieve the classification task. In a similar manner to Static SwiftR, it outputs dC_p, dD_p, dF_p, confidence, detection, and family probability distributions. Our RNN architecture is based on an LSTM model [117], which will be described, in details, in the next sections.

Training and Decision Making

Similar to Static SwiftR (Section 8.2.1), we train Dynamic SwiftR neural network on labeled behavioral reports (training set). Each report is labeled as described in Section 8.2.1. We leverage the serialized report as a word embedding sequence to provide the probability vectors dC_p, dD_p, dF_p, which are confidence, decision and family probability vectors. Dynamic SwiftR produces its final decision when $dC_p > dC_{th}$, where dC_{th} is our confidence threshold, even before the execution ends. For dD_p and dF_p, the decision, during the deployment process, is the same as in Static SwiftR (Section 8.2.1).

8.2.3 SwiftR Detection Strategy

SwiftR adopts two strategies to handle the outcome of its static and dynamic components. First, in the *parallel strategy*, SwiftR produces probability distributions for both components. We average the likelihood probabilities. The highest probability decides the class of the sample. Second, in the *serial strategy*, SwiftR produces outcomes from its static component. Based on the confidence in the probability distributions, under a certain threshold, we decide to invoke or not Dynamic SwiftR. The parallel strategy is convenient when there is a minimal overhead in producing behavioral reports. For example, in a production environment, the system runs the binary program anyway, and produces reports to check for abnormal behaviors.

8.3 Implicit Features

In this section, we describe the features used by SwiftR to fingerprint ransomware. We use the implicit terminology because the designed features are raw, and the neural network needs to learn explicit security features for fingerprinting purposes. We design the feature representation that retains the maximum information about the sample.

8.3.1 Implicit Static Features

We divide the static features into binary content and assembly content features.

Binary Content Features

In the binary content, we treat the binary sample surface as a sequence of bytes. The goal is to compute the bytes entropy of different regions of the binary sample.

Byte Entropy Matrix. The Byte Entropy Matrix (BEM) is a raw representation that summarizes the binary content of a given sample. We deal with a fixed-size format. As such, BEM is a 4096×4096 matrix, and we keep maximum information for the fingerprinting tasks. We need a small matrix because this representation will be the input to a Convolutional Neural Network (CNN). As presented in Algorithm 10, we commence the computation of BEM by dividing the binary sample into $2^{12} = 4096$ binary regions. We calculate byte 4-grams of each binary region and represent it as a fixed-size vector (2^{12}) using the feature-hashing technique [173]. We argue that the amount of information proportionally increases with the number of regions. For each region, we apply the feature-hashing technique (or hashing trick) [173], followed by L2 normalization on the computed byte 4-grams of a given region. The output is the byte entropy matrix, with a uniform size ($2^{12} \times 2^{12}$), for an arbitrary size sample. However, the size of BEM is still too large and it is sparse, which could influence the efficiency of the fingerprinting process.

Algorithm 10: Compute the Binary Entropy Matrix

Input : $BSeq$: Bytes Sequence
Output: BEM: Bytes Entropy Matrix

begin
  $BEM \leftarrow$ `Matrix(`*row=4096, col=4096*`)`;
  $BSlen \leftarrow size(BSeq)$;
  $RegSize \leftarrow size(BSeq)/4096$;
  foreach $idx \in [0..4096]$ **do**
    $rest \leftarrow size(BSeq[RegSize + idx, BSlen])$;
    if $rest >= RegSize$ **then**
      $BinReg \leftarrow BSeq[idx, RegSize + idx]$;
    else
      $BinReg \leftarrow BSeq[idx, BSlen]$;
    end
    $BinNGrams \leftarrow Get4grams(BinReg)$
    $ByteFreqVec \leftarrow$ `Vector(`*size=4096*`)`;
    $EntropyVec \leftarrow FHash(BinNGrams)$;
    $EntropyVec \leftarrow L2Norm(EntropyVec)$;
    $BEM[idx, :] \leftarrow EntropyVec$;
  end
  return BEM;
end

Compressed BEM. In this step, we compress further the BEM into an 256×256 matrix and obtain a Compressed BEM (CBEM). SwiftR simply leverages Principal Component Analysis (PCA) to reduce the size of the BEM to make the detection faster. Our approach relies on applying PCA to BEM in the training and deployment as depicted in Algorithm 11.

Algorithm 11: BEM Compression

```
Input  : BEM: Bytes Entropy Matrix
Output: CBEM: Compressed BEM
begin
    compress ← PCA(components=256);
    sBEM ← Reshape(BEM, (65536, 256));
    CBEM ← compress(sBEM);
    return CBEM;
end
```

CBEM Visualization: In order to show the relevance of using CBEM in ransomware fingerprinting, we visualize the CBEM of the samples from our dataset. Figure 8.3 depicts the samples' CBEM after applying the Whittaker-Shannon interpolation[1]. The first row (i.e., Figures 8.3(a), 8.3(b), 8.3(c)) shows the samples' CBEM from the benign dataset whose MD5 values start with a943d, 98687, and 8139f. Similarly, other malware are shown in the second row (i.e., Figures 8.3(d), 8.3(e), 8.3(f)), WannaCry samples are shown in the third row (i.e., Figures 8.3(g), 8.3(h), 8.3(i)), and TeslaCrypt samples are shown in Figures 8.3(j), 8.3(k), 8.3(l). The aforementioned figures show distinguishable patterns among the categories of samples. Visually, we are in a position to cluster the samples into categories. In particular, we clearly distinguish the WannaCry's and TeslaCrypt's patterns.

Assembly Intermediate Representation Content

In this section, we focus on the assembly content of the samples. The assembly is the result of the disassembly operation on the binary sample. Afterwards, we lift the architecture specific assembly to an intermediate representation. In our case, we choose VEX IR [35] due its simplicity and the availability of open-source tools to process it, such as Angr [34].

Assembly Feature Sequences. The process of extracting static features from assembly is presented in Figure 8.4. SwiftR disassembles the program into architecture-specific assembly. Next,

[1] http://bit.ly/2sPmlIp

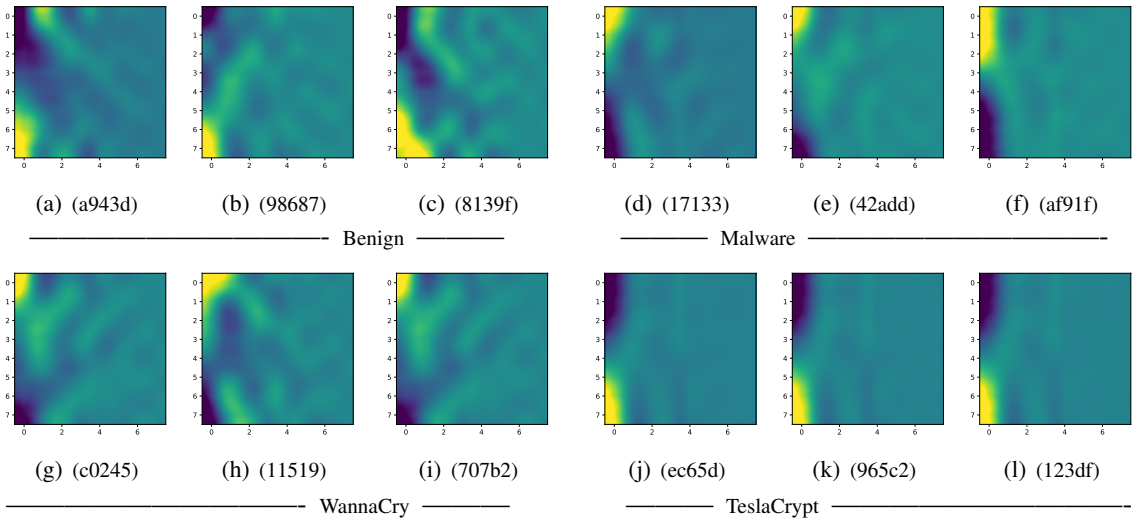


Figure 8.3: CBEM Visualization using Sinc Interpolation

we lift it to intermediate representation, which is architecture-independent. Finally, we extract feature sequences.

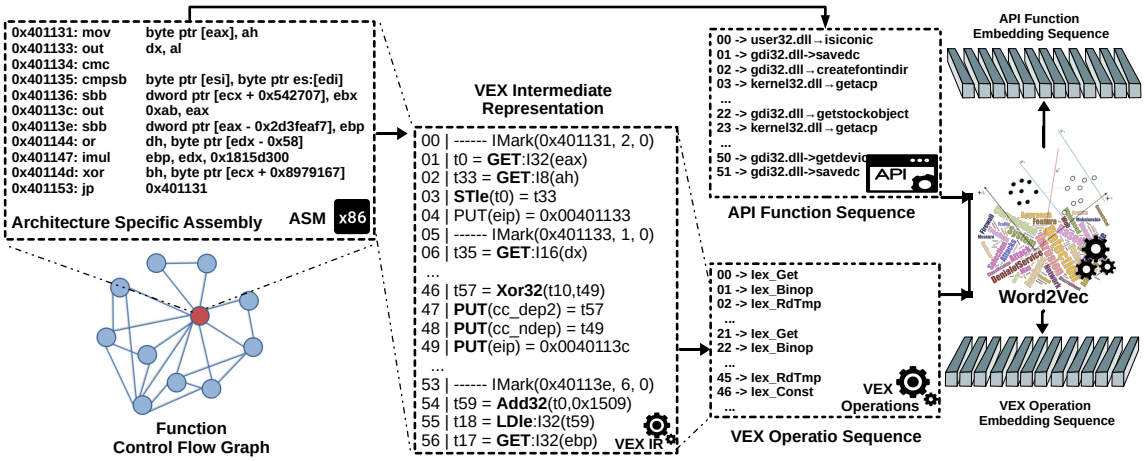


Figure 8.4: Implicit Static Feature Generation Process

VEX Operation Sequence: From VEX IR, SwiftR employs raw VEX operations as representation of program semantics. We argue that this raw sequence will preserve the essential information to fingerprint ransomware from other types. SwiftR puts the burden of automatic feature engineering on neural networks to discover granular features. As shown in Figure 8.4, we keep the order in the VEX operations sequence by concatenating operations of basic blocks at the function level.

Afterwards, we concatenate the sequences associated with functions to produce one sequence for the entire sample.

API Function Invocation Sequence: As shown in Figure 8.7, we leverage function invocations because they reflect the called Application Programming Interfaces (API) by a program during the execution. For example, the usage of cryptographic primitives is an essential part of crypto-ransomware. Similarly, we keep the order of function invocations sequence starting from the basic blocks to the functions, and finally the program.

Sequence Visualization: We visualize sequences to check the similarities between samples. Figure 8.6 depicts the discrete VEX operation (the operation is replaced with a unique identification number) and a sequence of a hundred operations in bar charts. The height of each bar represents the numerical identification of the VEX operation. It is noticeable that sequences are very similar in the case of WannaCry samples (i.e., Figures 8.6(g), 8.6(h), 8.6(i)), as well as in the case of TeslaCrypt samples (i.e., Figures 8.6(j), 8.6(k), 8.6(l). On the other hand, the sequences of benign samples are different. In the case of other malware (i.e., Figures 8.6(d), 8.6(e), 8.6(f)), the sample sequences are not similar but they have more common parts compared to the benign sequences (i.e., Figures 8.6(a), 8.6(b), 8.6(c)). Figure 8.5 shows the chart of a complete sequence that SwiftR will leverage to fingerprint ransomware.

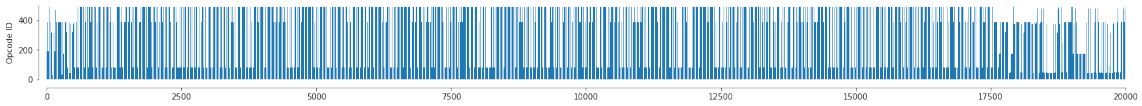
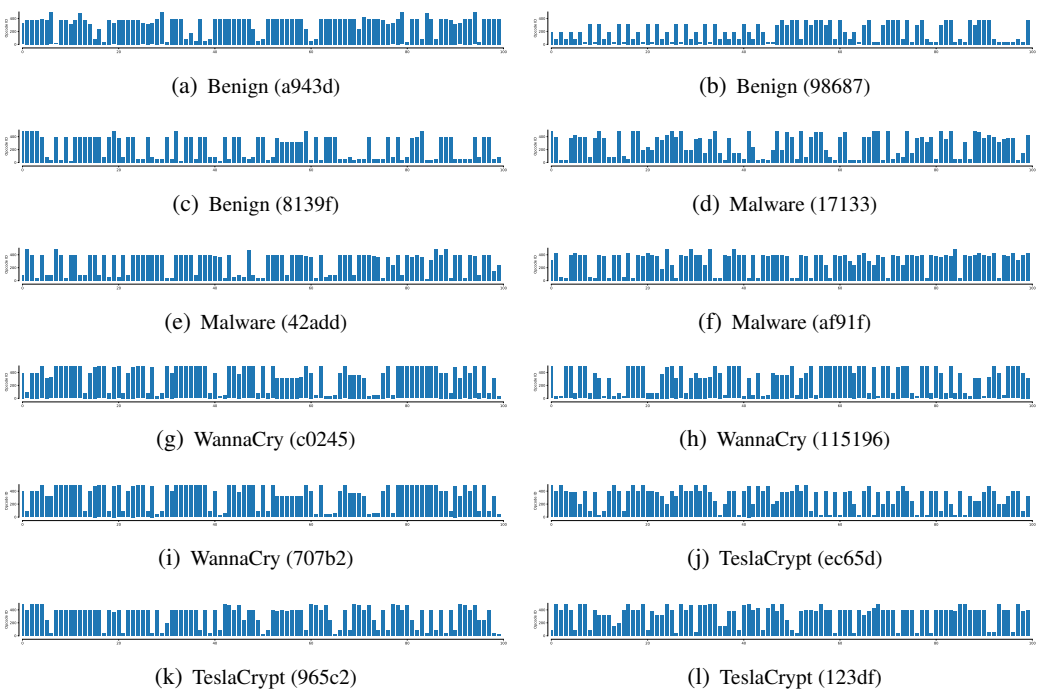


Figure 8.5: Full Opcode Sequence of TeslaCrypt (ec65d) Sample

Assembly Embedding Sequences. Word embedding techniques, such as word2vec [153] and GloVe [159], are extensively used in state-of-the-art natural language processing solutions to map words to embeddings. In cyber security, embedding techniques show high effectiveness in code identification. For example, Asm2Vec [100] uses an extension of word2vec embedding technique for function identification. The authors show the resiliency of embedding techniques to code obfuscation. Other proposals, such as [132, 193], employ embeddings for malware detection. SwiftR leverages word2vec [153] embedding technique to map feature sequences to embedding sequences.



(a) Benign (a943d) (b) Benign (98687)

(c) Benign (8139f) (d) Malware (17133)

(e) Malware (42add) (f) Malware (af91f)

(g) WannaCry (c0245) (h) WannaCry (115196)

(i) WannaCry (707b2) (j) TeslaCrypt (ec65d)

(k) TeslaCrypt (965c2) (l) TeslaCrypt (123df)

Figure 8.6: VEX Operation Sequences (100 Operations)

8.3.2 Implicit Dynamic Features

In this section, we detail the implicit dynamic features. The following properties represent the design goals of SwiftR implicit dynamic features: (1) agnostic to execution environment, (2) automatic feature engineering, and (3) minimal preprocessing. The extraction processing starts from a behavioral report in its textual format and ends with a behavioral embedding sequence. Figure 8.7 depicts the SwiftR process to generate implicit dynamic features.

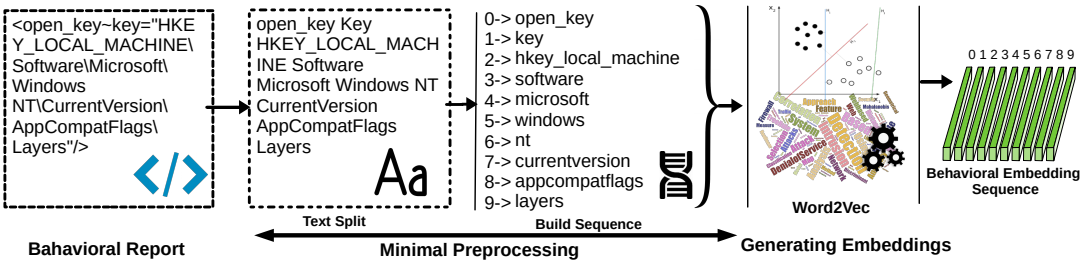


Figure 8.7: Implicit Dynamic Feature Generation Process

Sequence of Words (SoW)

SwiftR produces a sequence of words from a behavioral report with minimal preprocessing. We simply split the text of the report into words on spaces and special characters as shown in Figure 8.7. The result is a sequence of words that follows the order in the original report. In contrast with bag of words (BoW) [128], unique words may repeat in SoW model.

Behavioral Embeddings Sequence

Similarly to static sequences (Section 8.3.1), we use word2vec [153] to map sequences of words to embeddings. Each word is mapped to one embedding. The word embedding (word2vec) is trained separately in unsupervised manner as it will be described in Section 8.4.1.

8.3.3 Deep and Conventional Learning for Implicit Features

For automatic pattern recognition from the previous implicit features, SwiftR employs Neural Networks (NNs), such as Convolutional NNs (CNNs) and Recurrent NNs (RNNs). In contrast to Conventional Machine Learning (CML) techniques, such as Support Vector Machines (SVMs), SwiftR NN provides the following advantages: (i) *Automatic Feature Engineering*: Implicit or raw features need more processing to produce features for CML pipeline. This requires manual feature selection, which is time-consuming. In particular, to tackle the exponential growth of new malware with new features, which slows cycling through building models, SwiftR NN provides automatic feature engineering from raw feature representation. (ii) *Heterogeneous Content*: CML requires the merger and fusion of features from different contents into a single feature vector. In contrast, SwiftR NN keeps the raw features form the different contents which allows building generic models. (iii) *Temporal Information*: CML ignores the temporal and spatial relationship of our raw features by merging and selecting the feature. As shown in Figure 8.5, VEX IR operations' values are important as well as the order between the operations.

8.4 SwiftR Neural Network Architecture

In this section, we present the neural network architectures of SwiftR. First, we describe SwiftR word embedding model, which produces our implicit feature embeddings. Next, we present SwiftR neural network architectures and their training for static and dynamic features.

8.4.1 SwiftR Word Embedding

SwiftR word embedding model is the key component to produce our so-called implicit features. Word2vec (as presented in Chapter 7)employs a vector space model to represent a report's words in a continuous vector space where words with similar semantics are mapped closely in the space. From a security perspective, we want to map our features (word identifiers, VEX operations, function invocations, and behavioral report words) to continuous vectors where their semantics is translated into a distance in the vector space.

8.4.2 Static SwiftR Neural Network

In the previous sections, we discussed the generation of multiple implicit features: (1) binary CBEM, (2) VEX embedding sequence, and (3) function embedding sequence. Feature (1) is a 256×256 matrix, while Features (2) and (3) are sequences of embeddings. Due to the diversity of the content types and their feature representations, we build our specific neural network architecture since existing architectures do not fit our use case. We rely on existing neural network models, such as CNNs and MLPs (Multi-Layer Prceptron), as basic blocks to build SwiftR hierarchical architecture. The key idea is that each implicit feature has its NN model as presented in Figure 8.1. This is the first stage, which plays the role of the automatic feature engineering component. In order to learn from Feature (1), SwiftR employs the architecture detailed in Table 8.1, which is inspired by the VGG model [174], an important neural network structure for image classification. In order to learn from Feature (2) and Feature (3), we endow SwiftR with a CNN model that is inspired by the model proposed in [138], which is designed to learn from embedding sequences. Our sequence model is presented in Table 8.2.

The second stage plays the role of the decision making component, as depicted in Figure 8.1.

	#	*Layers*	*Options*
B1	1	Conv	Filter=128, Kernel=(3,3), Stride=(1,1), Zero-Padding, Activation=ReLU
	2	BNorm	BatchNormalization
	3	MaxPooling	Kernel=(2,2), Stride=(2,2), Zero-Padding
B2	4	Conv	Filter=128, Kernel=(3,3), Stride=(1,1), Zero-Padding, Activation=ReLU
	5	BNorm	BatchNormalization
	6	MaxPooling	Kernel=(2,2), Stride=(2,2), Zero-Padding
B3	7	Conv	Filter=128, Kernel=(3,3), Stride=(1,1), Zero-Padding, Activation=ReLU
	8	BNorm	BatchNormalization
	9	MaxPooling	Kernel=(2,2), Stride=(2,2), Zero-Padding
B4	10	MaxPooling	Global Max Pooling
	11	FC	#Output=512, Activation=ReLU
	12	BNorm	BatchNormalization

Table 8.1: CBFM Neural Network (CBFM NN)

	#	*Layers*	*Options*
B1	1	Conv	Filter=128, Kernel=(3,K), Stride=1, Zero-Padding, Activation=ReLU
	2	BNorm	BatchNormalization
	3	MaxPooling	Global Max Pooling
B2	4	FC	#Output=512, Activation=ReLU
	5	BNorm	BatchNormalization

Table 8.2: Sequence Neural Network

	#	*Layers*	*Options*
B1	1	FC	#Output=512 , Activation=ReLU
	2	BNorm	BatchNormalization
B2	3	FC	#Output=512, Activation=ReLU
	4	BNorm	BatchNormalization
B3	5	FC	#Output=512, Activation=ReLU
	6	BNorm	BatchNormalization
B4	7	FC	#Output=512, Activation=ReL
	8	BNorm	BatchNormalization
B5	9	FC	#Output=512, Activation=ReLU
	10	BNorm	BatchNormalization
B6	11	FC	#Neurons=$\{sC_p, sD_p, sF_p\}$, Activation=Sigmoid

[1] The number of malware families in the training dataset.

Table 8.3: Decision Neural Network (DNN)

The decision making NN is presented in Table 8.3. The composition of the aforementioned stages delineates the hierarchical NN architecture of Static SwiftR.

8.4.3 Dynamic SwiftR Neural Network

In previous sections, we discussed how SwiftR is producing behavioral embedding sequences from the textual content of behavioral reports. In order to recognize ransomware patterns from these raw sequences, we employ the neural network model illustrated in Table 8.4. The aforementioned

model is based on the supervised RNN model for ransomware fingerprinting. This model is inspired by advanced natural language processing systems, where LSTM (Long Short Term Memory) [117] is a core component. The intuition behind choosing LSTM, and recurrent neural networks, in general, is the aim to capture the information of sequential words in dynamic analysis reports. LSTM is known to learn from long sequences and retain information about the relation between the sequence items, even on relatively long sequences. This information allows to increase the accuracy of malware detection in the context of dynamic analysis.

In this context, σ is the sigmoid function, c_t is the cell state at time t, h_t is the hidden state at time t, h_{t-1} is the hidden state of the layer at time $t-1$ or the initial hidden state at time 0, and i_t, f_t, g_t, o_t are the input, forget, cell, and output gates, respectively [117].

	#	***Layers***	***Options***
B1	1	LSTM	Hidden=128
	2	Dropout=0.2	Dropout Regularization
B2	3	FC	#Neurons=$\{sC_p, sD_p, sF_p\}$, Activation=Sigmoid

Table 8.4: Dynamic SwiftR Neural Network

8.4.4 SwiftR Training

We abstract SwiftR neural network architectures to the following functions:

$$StaticNN\left(x_{static}\right) = \grave{y} \tag{27}$$

$$DynamicNN\left(x_{dyn}\right) = \grave{y} \tag{28}$$

Where $\grave{y}$ is the output based on a fingerprinting task. The training dataset is a list of tuples $\langle (x_0, y_0), (x_1, y_1), \cdots, (x_m, y_m) \rangle$, where $x_i \rightarrow \langle vex_i, api_i, CBEM_i, report_i \rangle$ and y_i is a label. The training objective is to make $\grave{(}y_i) \approx y_i$ and $\grave{(}y_i) \approx y_i$ for our training dataset and the aim is to generalize to other examples outside the training dataset. Formally, we define our **objective functions**, $J_{static}(\theta_{static})$ and $J_{dynamic}(\theta_{dynamic})$, with θ_{static} and θ_{dyn} being the parameters to learn using gradient-based optimization algorithm [113]:

$$J(\theta_{static}) = \frac{1}{2m}\sum_{i=1}^{m}\left(StaticNN\left(x_i;\theta(x_i)\right) - y_i\right)^2 \tag{29}$$

$$J(\theta_{dyn}) = \frac{1}{2m}\sum_{i=1}^{m}(DynNN(x_i;\theta(x_i)) - y_i)^2 \tag{30}$$

To this end, we search for θ parameters that minimize objective functions $J(\theta)$. For this reason, we leverage the gradient descent optimization algorithm [113] to find the arguments of the minimal $J(\theta)$ by iteratively updating θ until convergence:

$$\theta_j = \theta_j - \alpha\frac{\partial J(\theta)}{\partial\theta} \tag{31}$$

In Equation 31, α is the learning rate. In all our training, we use $\alpha = 3E-4$ from Adam [139], an adaptive learning rate for gradient descent optimization algorithm. The chosen learning rate comes from different experiments.

8.5 Evaluation Dataset

The evaluation dataset is composed of ransomware, benign applications, and general malware samples. The total size of the evaluation dataset is 41.3k samples collected over the last six years from the different categories, as shown in Table 8.5.

Category	**# Samples**	**Date**
Ransomware	10.3k	05-2012 to 09-2017
Bengin	10k	09-2011 to 11-2017
Malware	20k	01-2014 to 05-2017
Total	40.3k	09-2011 to 11-2017

Table 8.5: Evaluation Dataset

The ransomware evaluation dataset is the backbone of our dataset. It is composed of twelve ransomware families with 10.3k samples in total, as depicted in Table 8.7. The process of collecting the dataset starts from RansomwareTracker [2] from which we get the malware hashes along with

[2]https://ransomwaretracker.abuse.ch

their ransomware families. The families and hash mappings have been confirmed and corrected manually for some samples using reports from VirusTotal [3]. The latter provides the family names given by the vendors. Furthermore, we leverage VirusTotal to download the actual samples. The diversity of ransomware families increases the robustness of the evaluation process. In addition to family diversity, our dataset of ransomware samples is temporally distributed over the period 2012 to 2017, as shown in Figure 8.8. Our time ground-truth is based on the VirusTotal first date analysis. The time distribution is another factor that is used to enhance the evaluation.

	Family	#Sample		Family	#		Family	#		Family	#
01	**allaple**	4506	19	**madang**	144	37	**gepys**	47	55	**resdro**	23
02	**virut**	3676	20	**chir**	126	38	**buzus**	46	56	**sirefef**	22
03	**nabucur**	2005	21	**delf**	123	39	**picsys**	45	57	**waledac**	22
04	**mira**	1376	22	**zbot**	120	40	**jeefo**	44	58	**fakeav**	21
05	**sality**	1043	23	**fynloski**	111	41	**kuluoz**	44	59	**koutodoor**	20
06	**sivis**	1009	24	**matsnu**	110	42	**kolabc**	39	60	**tempedreve**	18
07	**shodi**	932	25	**reveton**	100	43	**zaccess**	36	61	**mabezat**	17
08	**ramnit**	496	26	**kryptik**	95	44	**encpk**	35	62	**looked**	17
09	**luiha**	368	27	**cutwail**	82	45	**pondfull**	34	63	**magania**	15
10	**vobfus**	332	28	**fasong**	82	46	**fesber**	33	64	**winwebsec**	15
11	**ipamor**	320	29	**daws**	80	47	**waski**	31	65	**viking**	14
12	**loadmoney**	317	30	**urelas**	74	48	**rebhip**	29	66	**tufik**	12
13	**upatre**	237	31	**swrort**	73	49	**slugin**	29	67	**nuqel**	12
14	**bayrob**	211	32	**skyper**	70	50	**zegost**	29	68	**simbot**	12
15	**neshta**	172	33	**expiro**	62	51	**ardamax**	28	69	**geral**	11
16	**unruy**	166	34	**jadtre**	58	52	**ircbot**	28	70	**wauchos**	7
17	**autoit**	159	35	**nitol**	52	53	**wonton**	28	71	**blakamba**	7
18	**valla**	158	36	**alman**	50	54	**nimnul**	23	72	**floxif**	7
					Total	**20000**					

Table 8.6: Malware Dataset Families Distribution

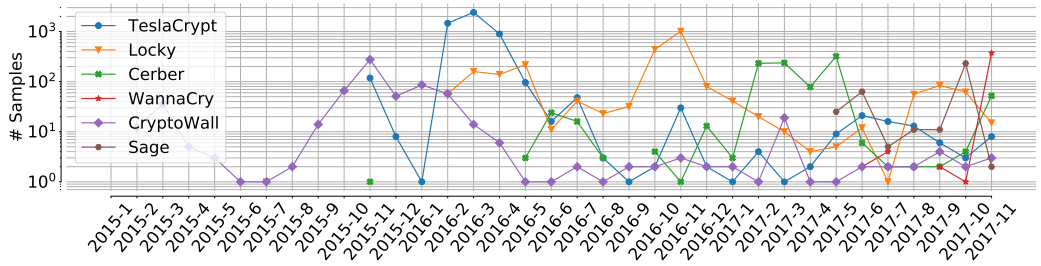


Figure 8.8: Ransomware Dataset Families Distribution Overtime

The benign dataset needs to represent various categories of benign applications in terms of app usage. The latter refers to the purpose of the application such as media, utilities, web browsers, etc. For the above factors, we leverage Ninite[4], an automatic app installation to setup all categories of

[3]https://www.virustotal.com
[4]https://ninite.com/

#	Family	#Sample	#	Family	#Sample
01	**teslacrypt**	5185	07	**ctb-locker**	83
02	**locky**	2522	08	**torrentloc**	43
03	**cerber**	1006	09	**paycrypt**	22
04	**cryptowall**	690	10	**dmalocker**	8
05	**wannacry**	383	11	**petya**	3
06	**sage**	348	12	**padcrypt**	1
		Total	10294		

Table 8.7: Ransomware Families Distribution

apps form a Firefox web browser to a Blender 3D modeling tool. The collection ends up with 10k benign samples distributed in terms of application and time, as presented in Table 8.5.

The general malware dataset contains 20k malware samples that are collected from over 70 general malware families, as presented in Table 8.6. The family-related ground-truth is provided by a third-party security vendor (ThreatTrack Security[5]). We explicitly choose to have over 70 malware families in the malware dataset to reflect the diverse families of malware.

8.6 Effectiveness Evaluation

In this section, we report on the effectiveness results of SwiftR. We assess the effectiveness of SwiftR framework under different cross validation settings. We consider three evaluation metrics: (1) *F1-Score*, (2) *Area Under the ROC curve (AUC)*, and (3) *False positive rate (FPR)*.

8.6.1 Evaluation Process

We use K-fold cross-validation to validate the detection results; in particular, we employ 10, 5, and 3-fold cross-validation. The reason behind using multiple $K \in \{10, 5, 3\}$, is to check SwiftR performance under different dataset scales and to provide more confidence. For a given K-fold, we divide the evaluation dataset into K portions; $K - 1$ portions are for the training, and the last one is for the testing. We repeat this process for K times. The final performance result is the average of the K results. We repeat the experiment ten times for each K-fold setting. The final results represent average and standard deviation values of the individual experimentation across the K-folds. Furthermore, the evaluation dataset as a whole will serve as the training dataset for SwiftR models in the production case study (Section 8.7).

[5]https://tinyurl.com/ydaovjkz

8.6.2 Evaluation Checklist

We aim to answer the following questions: (1) How accurately can SwiftR distinguish ransomware from benign and malware samples, and attribute such samples to a known ransomware family? (2) Can SwiftR framework be generalized to detect temporal unknown samples and unknown families? To answer the above questions, we arrange SwiftR evaluation into the following evaluation tasks:

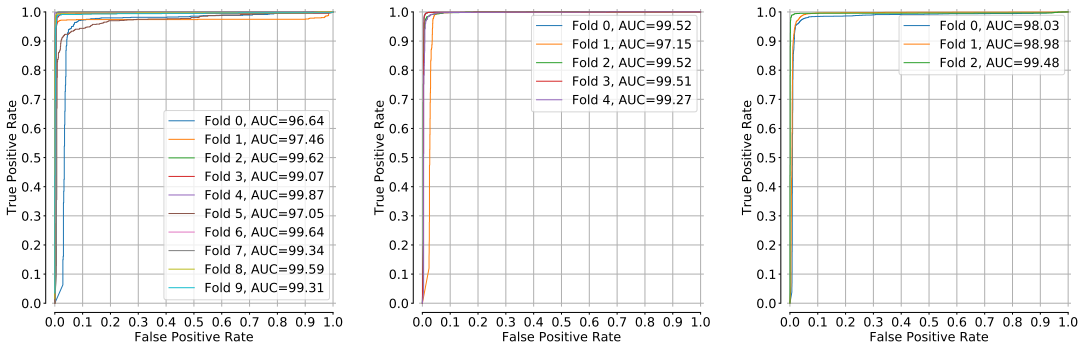
- *Ransomware Detection:* This is the task in which SwiftR differentiates between benign and ransomware samples (Section 8.6.3).
- *Ransomware Attribution:* This is the task where we assess the ransomware attribution performance of SwiftR in recognizing ransomware from other malware samples (Section 8.6.4).
- *Ransomware Family Attribution:* In this task, we measure the ability of SwiftR to attribute a given ransomware sample to its actual ransomware family (Section 8.6.5).
- *Detection of Unknown Family:* This task answers whether SwiftR detection task could distinguish ransomware samples from an unknown ransomware family, and from benign samples even though this family is not part of the training dataset (Section 8.6.6).
- *Detection of Unknown Sample (Time):* In this task, we check the detection performance of unknown ransomware by taking into account the temporal distribution of the samples. As such, we train using samples from year x and assess the detection performance of samples from year $x + 1$ (Section 8.6.6).

8.6.3 Ransomware Detection

In this section, we focus on the detection performance of SwiftR. Basically, given a binary program, SwiftR detects whether it is ransomware or benign application.

# Fold	F1-Score(%)	AUC (%)	FPR (%)
10	98.75 ± 1.43	99.23 ± 1.56	0.88 ± 0.27
5	97.54 ± 2.33	98.85 ± 1.69	1.15 ± 0.65
3	97.59 ± 1.38	98.89 ± 0.71	1.74 ± 0.87

Table 8.8: Detection Results Summary



(a) 10-Fold Cross-Validation (b) 5-Fold Cross-Validation (c) 10-Fold Cross-Validation

Figure 8.9: Detection ROC Curves

The detection results of SwiftR are summarized in Table 8.8. In each cross-validation setting, Table 8.8 shows the average F1-score (threshold 0.5), AUC (Area Under the Curve), and FPR (False Positive Rate) along with the standard deviation. As presented in Table 8.8, SwiftR shows high detection performance of SwiftR with low FPR.

In this section, we answer the question related to the detection performance. We provide a complete answer for the detection capability of SwiftR as part of a production test, as presented in Section 8.7.

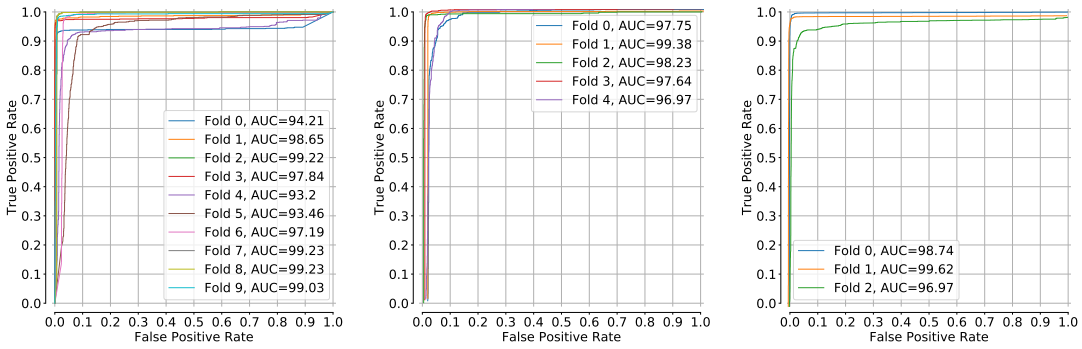
8.6.4 Ransomware Attribution

In this section, we assess the performance of SwiftR on segregating ransomware from general malware. In essence, given a malware sample, SwiftR discriminates if it is ransomware or general malware. We assume in this use case that SwiftR is deployed on a filtering environment where there is more general malware than ransomware. In this context, the attribution capability of SwiftR helps to filter ransomware samples.

# Fold	F1 (%)	AUC (%)	FPR (%)
10	96.72 ± 01.56	97.22 ± 01.75	2.11 ± 01.24
5	96.98 ± 01.15	97.89 ± 02.83	2.02 ± 01.56
3	96.11 ± 01.33	96.89 ± 01.15	2.49 ± 01.45

Table 8.9: Attribution Results Summary

Table 8.9 depicts a summary of the attribution performance results under the different cross-validation settings in terms of F1-score, false positive rate (FPR) (threshold $t = 0.5$), and AUC



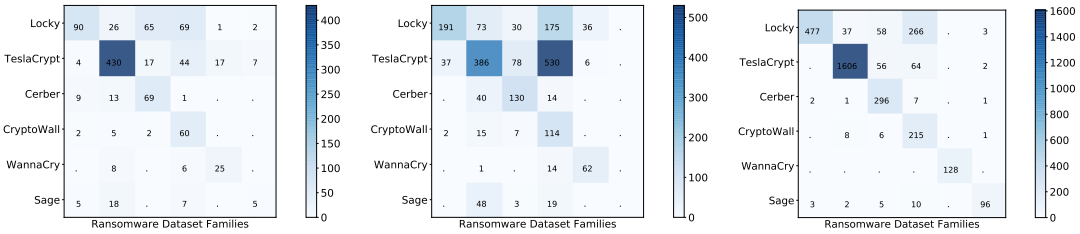
(a) 10-Fold Cross-Validation (b) 5-Fold Cross-Validation (c) 10-Fold Cross-Validation

Figure 8.10: Attribution ROC Curves

averages and their standard deviations. As shown in Table 8.9, SwiftR has a high performance in the attribution task. However, one can note that SwiftR attribution performance is slightly less than its detection performance results. In other words, SwiftR can better segregate between ransomware and benign samples than it does between general malware and ransomware samples. There is a logic behind this observation since general malware and ransomware share maliciousness behaviors, which are translated into a static content and dynamic behaviors. The shared maliciousness makes SwiftR attribution more difficult task than the detection where there is little sharing between benign and ransomware samples. In the next section, we check the effectiveness of SwiftR in ransomware family attribution.

8.6.5 Ransomware Family Attribution

In this section, we evaluate SwiftR with respect to the ransomware family attribution task, which is related to the other tasks of SwiftR (Section 8.6.3 and 8.6.4). In other words, after SwiftR discovers a given sample to be ransomware, we are looking for its ransomware family. In contrast to previous tasks, SwiftR family attribution has many decision outputs according to the number of targeted families. In our case, we have six families (Locky, TeslaCrypt, Cerber, CryptoWall, WannaCry, Sage) from the top ones in our evaluation dataset. We gather the rest of the samples in a separate category, named *Unknown*. In machine learning terminology, the detection tasks represent a binary classification problem (only two classes); the family attribution task is a multi-classification problem. The problem of ransomware family attribution is harder than the previous tasks: (i) There



(a) 10-Fold Cross-Validation (b) 5-Fold Cross-Validation (c) 10-Fold Cross-Validation

Figure 8.11: Family Attribution Confusion Matrices

are many ransomware families to consider. (ii) Compared to attribution (Section 8.6.4), the similarity between samples of different ransomware families is quite high, which makes the segregation task more challenging. (iii) It is hard to consider unknown families, beyond the ones in the training dataset.

# Fold	F1-Score (%)	Precision (%)	Recall (%)
10	93.58 ± 02.46	94.94 ± 03.66	92.73 ± 05.56
5	94.51 ± 03.24	95.47 ± 02.00	93.79 ± 01.22
3	92.91 ± 03.85	95.16 ± 05.06	91.60 ± 07.63

Table 8.10: Family Attribution Results Summary

Table 8.10 summarizes the performance results of SwiftR family attribution under the different K-fold cross-validation setups, $K \in \{10, 5, 3\}$. As shown in Figure 8.10, SwiftR displays a high performance under different evaluation settings; the F1-score is above 90% in the 10, 5, and 3-fold cross-validation setups. Due to the difficulty of the task, we consider these results as very good because the baseline model (random decision) has about 17% (100% / 7) (number of families + Unknown Class), but we obtain about 94%. The detailed results are presented as confusion matrices in Figures 8.11(a), 8.11(b), 8.11(c). In the next section, we discuss the performance of SwiftR on time-based and family-based unknown samples.

8.6.6 Unknown Samples Fingerprinting

In this section, we present the evaluation results of SwiftR against unknown samples. Here, we address two particular cases of unknown samples, namely: (i) time-based unknown samples and (ii) family-based unknown samples, as will be presented in the rest of the section.

Time Scenario

In this scenario, we evaluate the resiliency of SwiftR against time changes. In other words, we check the time resiliency of SwiftR detection and attribution. We formulate this scenario based on the following question: *How good will be the performance of* SwiftR *in the next year if we train it on samples of the current year?*

Setup. The setup process starts by dividing the evaluation dataset into training and testing sets, based on the timestamps of the samples. For malware/ransomware, the timestamps represent the first detection time for a given malware or ransomware sample. Our ground truth comes from VirusTotal reports by considering the *first seen* field. For benign samples, we rely on the release date of the software. This results in a training set of samples only from 2016 (ransomware, malware, and benign samples) while the testing set consists of samples only from 2017; where we ignore samples of other years due to a severe unbalancing.

Results. Figure 8.12 shows the performance of the SwiftR models trained on 2016 dataset and tested against the 2017 dataset. The figure shows the performance of SwiftR in the attribution task, where it achieves an average of 92% F1-score for many experiments. Attribution performance is comparable to the results presented in Section 8.6.4. On the other hand, SwiftR obtains 93% F1-score in the detection task, which is also similar to the results presented in Section 8.6.3. Overall, SwiftR performs well and shows its time-resiliency.

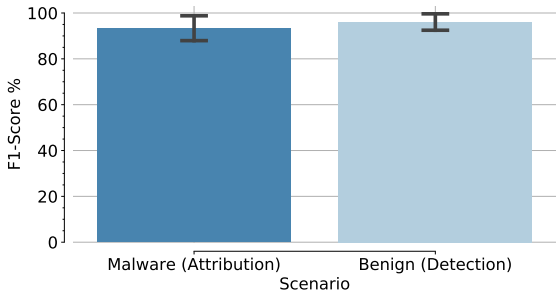


Figure 8.12: Time Resiliency Results

Family Scenario

In this scenario, we check the resiliency of SwiftR against new ransomware family. In particular, we address the following question: *How would* SwiftR *perform against samples from unknown ransomware families when we train it on other families, in the case of the detection and the attribution tasks?*

Setup. The setup of this evaluation involves splitting the ransomware dataset by family as presented in Table 8.7. We evaluate each ransomware family separately. Thus, we train the model on all the evaluation datasets for a given task, and we exclude one ransomware family samples, which we use for the testing and reporting of the effectiveness of SwiftR on unknown ransomware families. We conduct the training/testing for all top six ransomware families (TeslaCrypt, Locky, Cerber, CryptoWall, WannaCry, Sage) for detection and attribution tasks.

Results. Figure 8.13 presents the average detection performance in terms of F1-score, along with its standard deviation, under threshold $t = 0.5$. SwiftR shows a high detection performance for most of the ransomware families by achieving over 90% F1-score. The exception was WannaCry family, where SwiftR achieves 82% F1-score. Also, the error bar in Figure 8.13 is tight for most families' evaluation, which corresponds to very similar F1-scores for most experiments. Figure 8.14 shows the attribution performance with respect to the F1-score. Compared to the detection task, SwiftR attribution has lower F1-score results. Similarly, the exception is WannaCry family where SwiftR achieves 82% F1-score.

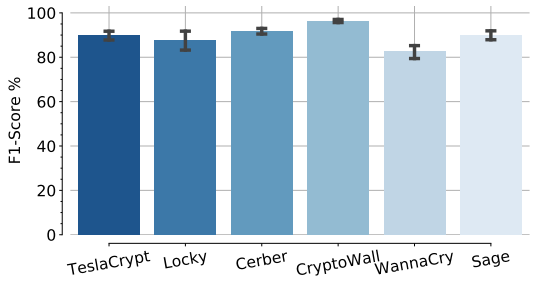


Figure 8.13: Ransomware Detection Result Unknown Family

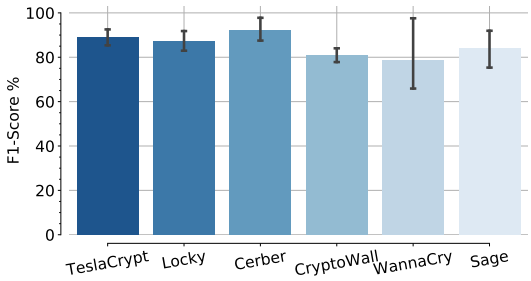


Figure 8.14: Malware Attribution Result Unknown Family

8.6.7 Efficiency Evaluation

In this section, we report the efficiency results of SwiftR in terms of runtime.

Setup

The efficiency evaluation setup starts by randomly selecting 500 samples from the evaluation dataset. We divide SwiftR process pipeline into three phases: (i) *Disassembly/Lifting*: SwiftR produces the assembly of the sample. We use Angr [34] for disassembly and lifting. Dynamic analysis runtime is not reported because it is constant (about 10 minutes). (ii) *Implicit Features Representation Phase*: SwiftR produces static and dynamic features and embedding sequences. (iii) *Decision Phase*: SwiftR decides on a given sample. In this evaluation, we measure the runtime for each sample for each phase separately and for each hardware. Notice that runtime measurement is performed only for a single thread running on the CPU.

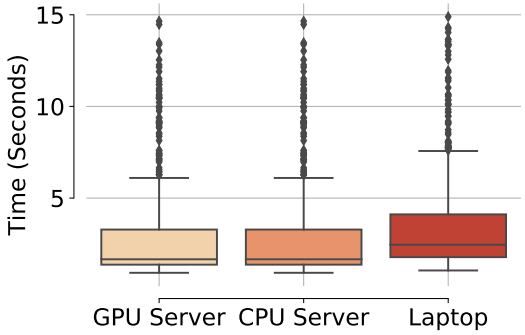


Figure 8.15: Disassembly and Lifting Runtime

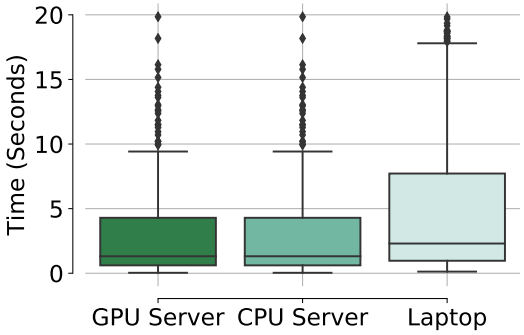


Figure 8.16: Raw Feature Representation Runtime

Results

Figure 8.15, 8.16, and 8.17 depict the efficiency results of SwiftR framework for the disassembly/lifting, feature representation, and decision phases respectively. Although there is an enormous difference between the evaluation machines regarding the specifications, their results are comparable for the disassembly and the feature representation phases. There is a logic behind this result with respect to the CPU and GPU server since the previous phases do not rely on GPU for computation and the servers have identical specification excluding the GPU. For the server, the maximum

Phase	GPU Server	CPU Server	Laptop
ASM/Lift	3.454 ± 4.591s	3.454 ± 4.591s	5.196 ± 7.045s
Raw Feature	3.018 ± 4.032s	3.018 ± 4.032s	6.658 ± 9.519s
Decision	0.004 ± 0.001s	0.017 ± 0.003s	0.373 ± 0.047s
Runtime	6.475 ± 8.624s	6.488 ± 8.626s	12.22 ± 16.62s

Table 8.11: Runtime in the Different Analysis Phases

(minimum) time for the disassembly and the representation was 60 and 38 (0.92 and 0.03) seconds respectively. However, there is a difference in the decision time, where the GPU option in the server is about four times faster than the CPU option.

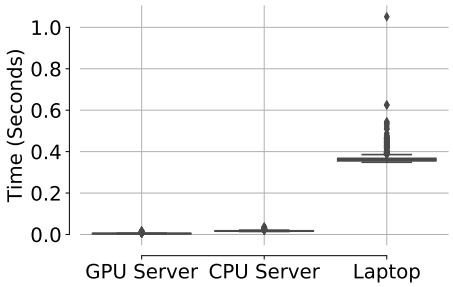


Figure 8.17: Decision Runtime

In the laptop machine, the disassembly and the representation phases show a comparable result to the servers because the servers are only using a single thread/core implementation. The difference will be more significant if SwiftR leverages all of the server's resources. Table 8.11 shows the average efficiency results of the SwiftR phases on the different machines.

8.7 Production Case Study

The evaluation in the previous sections (Section 8.6 and 8.6.7) shows the effectiveness and the efficiency of SwiftR in various fingerprinting tasks. However, in this section, we deploy SwiftR in a production setting to be evaluated against a large number of wild samples. This section reports the production performance of SwiftR at detection and attribution task. First, we describe the production setup and how we collected the performance results (Section 8.7.2 and 8.7.1). Second, we present the samples collected during production. Finally, we report the production results.

8.7.1 Setup

The overall process of the production starts by building the neural network model set of SwiftR. We train our neural networks on the whole evaluation dataset to produce deployment-ready models for SwiftR tasks, where we keep traces of the training performance. We measure the performance serially as we receive samples from various sources. We train multiple SwiftR instances to check the variety of results.

8.7.2 Wild Samples

During this case study period (six months), we progressively received as a feed from Threat-Track Security and other sources, a large number of samples of ransomware, benign binaries, and general malware. Table 8.13 summaries the amount of samples in each category.

8.7.3 Results

We report the production results in terms of F1-score, FPR, and AUC metrics (average $\pm$ standard deviation). We first note that the overall performance decreases in production compared to the evaluation (Section 8.6.3) in both fingerprinting tasks. In the detection task, SwiftR achieves $93.66 \pm 4.76\%$ F1-score on the case study samples; whereas in the evaluation it reaches $97.75 \pm 2\%$ F1-score (10-fold cross-validation). In other words, the performance drops by about 4% in F1-score metric. Interestingly, SwiftR detection has a small false positive rate $0.99 \pm 0.34\%$, and it is less than the one reported in the evaluation $1.00 \pm .75\%$. On the other hand, the SwiftR attribution achieves $91.81 \pm 3.25\%$ F1-score whereas the performance in the evaluation reached $94.72 \pm 04.56\%$ F1-score (10-fold cross-validation); thus, it drops by about 3% in F1-score. Besides, the false positive rate in the case study is $1.63 \pm 3.33\%$, which is surprisingly lower than the evaluation one of $2.44 \pm 1.84\%$. To sum up, the case study shows that the overall performance slightly drops, but the performance is still acceptable.

# Scenario	F1-Score(%)	FPR(%)	AUC(%)
Detection	93.66 ± 4.76	1.93 ± 3.34	95.17 ± 5.64
Attribution	91.81 ± 3.25	1.63 ± 3.33	92.67 ± 8.50

Table 8.12: SwiftR Performance during Study Case

Category	# Samples
Ransomware	38k
Benign	45k
Malware	100k
Total	183k

Table 8.13: Production Samples Statistics

8.8 Summary

In this chapter, we presented a novel framework for ransomware fingerprinting named SwiftR. To the best of our knowledge, SwiftR is the first automatic ransomware detection solution that relies on hybrid analysis, and that leverages deep learning techniques for ransomware mitigation. SwiftR relies on novel implicit features as input to the SwiftR neural networks for automatic feature engineering during decision making. SwiftR utility comes from its various usage scenarios: (i) Ransomware detection: Here SwiftR discriminates ransomware from the benign sample, the corresponding evaluation showing that SwiftR achieves 98% F1-score. (ii) Ransomware attribution: Here, SwiftR aims to distinguish between ransomware and general malware; it achieves a 96% F1-score. (iii) Ransomware family attribution: SwiftR attributes a given ransomware sample to its family; in this setting, it achieves a 95% F1-score. SwiftR prototype has been deployed for production usage where it was tested on more than 183k samples of general malware, benign, and ransomware samples. In the production case study, SwiftR showed a good performance with some drop compared to the evaluation results.

Chapter 9

Conclusion

9.1 Concluding Remarks

Software applications are at the core of many everyday life-depending services across a broad spectrum of devices, from mobile phones to transportation and medical equipment. At the heart of the rapid growth in software technologies, the development of mobile apps enhances both economic and social interactions. Mobile apps running on smart devices are nowadays ubiquitous due to their convenience. For instance, users can presently use apps as Google pay service to purchase products online and to make payments in retail stores. However, the growth of the mobile market apps has increased the concerns about the security of the apps. Android [27] is widely adopted mobile OS in smart devices, especially in the emerging Internet of Things (IoT) world through Android Thing [58], an Android-based IoT system.

Unfortunately, significant amounts of malicious software or malware, which are developed for a variety of purposes, aim at disrupting the well being of existing systems across many software platforms and hardware architectures. For example, $1,548,129$ and $2,333,777$ new Android malware were discovered [6] in 2014 and respectively 2015. Nowadays, the number of malware samples reaches into millions per month, and it is growing exponentially over time.

In this context, it is a desideratum to elaborate a scalable, robust, and accurate framework that tackles two specific problems: (i) Malware detection - distinguishing malicious from benign applications, and (ii) malware family attribution - assigning malware samples to known families.

This thesis is dedicated to tackling Android malware fingerprinting, detection and family attribution, at a large scale by proposing a series of frameworks and techniques to detect and attribute Android malware samples. Android malware detection was the main objective of our elaborated frameworks and systems. However, the core techniques and methods employed by these systems have potential application to general malware fingerprinting. The elaborated frameworks demonstrated very competitive Android malware fingerprinting results surpassing state-of-the-art solutions available at the time of writing this thesis. More specifically, we have presented the following contributions in this thesis:

- APK-DNA (Chapter 3): We elaborated a versatile approximate fingerprint [133] to capture the maximum information from the static content of an Android malware sample.
- Cypider (Chapter 3): We proposed a scalable malware clustering technique [130] leveraging APK-DNA approximate fingerprints along with graph partitioning techniques.
- DySign (Chapter 4): We designed and developed a platform-agnostic Android malware fingerprinting approach [128, 129] based on natural language processing techniques and dynamic analysis reports.
- MalDy (Chapter 4): We elaborated a supervised machine learning approach [125, 127] for malware detection on top of DySign fingerprints using dynamic analysis features.
- ToGather (Chapter 5): We proposed a cyber-infrastructure detector [124, 126] for Android malware in the cyber-space starting from network information of Android malware as well as static and dynamic analyses.
- MalDozer (Chapter 6): We proposed a portable and automatic Android malware detection and family attribution framework [69, 131, 132] that relies on sequences classification using deep learning techniques.
- PetaDroid (Chapter 7): We proposed a novel framework for Android malware detection that enhances the resiliency to code transformation and common obfuscation methods by input randomization. Also, we leveraged confidence-based detection to build new machine learning detection models that are able to adapt to new benign and malicious apps.

- SwiftR (Chapter 8): We proposed a ransomware fingerprinting framework based on intermediate code representation and demonstrated that SwiftR achieves high detection rates on different ransomware families.

9.2 Learned Lessons

We summarize in the following the main lessons that have been learned in this thesis:

- *Representation learning enables scalability*: Throughout this thesis, we learned that representation learning is at the core of malware automatic feature engineering. In our context, representation learning is an automatic and a data-driven process for generating malware embeddings. For example, the use of the word embedding (word2vec) technique helps learning the underlying semantics in an unsupervised manner. Existing large malware corpuses and unsupervised representation learning techniques help generating precise malware embeddings. More importantly, the precision of embeddings increases with the size of the malware corpus.

- *Natural language processing is a key*: In this thesis, we learned the usefulness of leveraging NLP abstractions in malware code analysis. Virtually, most NLP techniques used to segregate and analyze natural language are usable in the context of malware fingerprinting and detection. We believe that NLP abstractions and techniques are key for modern malware detection.

- *Machine learning is crucial*: We learned in this thesis that machine learning techniques are essential in elaborating advanced malware detection solutions. All existing state-of-the-art malware detection solutions rely nowadays on machine learning as a workhorse to fingerprint malware. Deep learning techniques have the edge over classical ones due to automatic discovery and filtering of relevant malware features from raw malware content.

- *Obfuscation learning is possible*: Throughout this thesis, we discovered the possibility to automatically learn obfuscation pattern through supervised techniques. Providing obfuscated malware samples to the training process allows learning common obfuscation methods while improving the overall generalization of the produced models.

9.3 Future Research Directions

In the following, we discuss potential future research directions:

- *Obfuscation on other platforms*: Throughout this thesis, we evaluate the proposed Android malware detection frameworks on different obfuscation techniques. Our frameworks show high detection performance on obfuscated Android malware. However, there is a need to evaluat our obfuscation resilient techniques on other platforms' obfuscated samples.

- *Tackling advanced obfuscations techniques*: In the context of this thesis, we carry out our evaluations on common obfuscation and code transformation techniques. As a future research direction, we could investigate the robustness of the proposed frameworks and techniques on advanced obfuscation techniques that employ heavy code transformation such as *control flow flattening*.

- *Additional deep learning techniques*: In this thesis, we employ different machine/deep learning techniques to fingerprint and detect Android malware. Exploring additional deep learning techniques can provide an important future direction for malware detection in general.

- *Network traffic features*: Throughout this thesis, we mainly employ dynamic and static analyses features to fingerprint Android malware. As a future research direction, we aim at engaging network traffic inspection as another source of features for Android malware detection.

Bibliography

[1] Android Auto - https://www.android.com/auto/, Accessed on April 2016.

[2] Android Things on the Intel Edison board - https://tinyurl.com/gl9gglk, Accessed on April 2016.

[3] Beware! New Android Malware Infected 2 Million Google Play Store Users - http://thehackernews.com/2017/04/android-malware-playstore.html, Accessed on April 2017.

[4] Bashlite family of malware infects 1 million iot devices - https://threatpost.com/bashlite-family-of-malware-infects-1-million-iot-devices/120230/, Accessed on August 2016.

[5] Virusshare malware repository- https://virusshare.com/, Accessed on August 2018.

[6] G DATA Mobile Malware Report - https://public.gdatasoftware.com/Presse/Publikationen/Malware_Reports/US/G_DATA_MobileMWR_Q4_2015_US.pdf, Accessed on December 2016.

[7] Open Handset Alliance - https://www.openhandsetalliance.com/android_overview.html, Accessed on December 2016.

[8] 360K New Malware Samples Hit the Scene Every Day - https://www.infosecurity-magazine.com/news/360k-new-malware-samples-every-day/, Accessed on December 2017.

[9] Lab detects 360,000 new malicious files daily - https://www.dailytrust.com.ng/lab-detects-360-000-new-malicious-files-daily.html, Accessed on December 2017.

[10] List: 44 Android apps infected with malware made their way to the Google Play store - http://clark.com/technology/google-play-malware-app-hummingbad, Accessed on December 2017.

[11] RaspberryPI 3 - https://www.raspberrypi.org/products/raspberry-pi-3-model-b/, Accessed on December 2017.

[12] Android Emulator - https://developer.android.com/studio/run/emulator, Accessed on February 2016.

[13] Andr/OpFake-U malware family - https://www.sophos.com/en-us/threat-center/threat-analyses/viruses-and-spyware/Andr~OpFake-U.aspx, Accessed on February 2016.

[14] As Ransomware Crisis Explodes, Hollywood Hospital Coughs Up 17,000 USD In Bitcoin - https://www.forbes.com/sites/thomasbrewster/2016/02/18/ransomware-hollywood-payment-locky-menace/#5526ad72408f, Accessed on February 2016.

[15] An Analysis of the WannaCry Ransomware Outbreak - https://securingtomorrow.mcafee.com/executive-perspectives/analysis-wannacry-ransomware-outbreak/, Accessed on February 2017.

[16] Have we reached the other side of the Bad Rabbit hole? - https://www.itproportal.com/features/six-months-on-have-we-reached-the-other-\side-of-the-bad-rabbit-hole/, Accessed on February 2018.

[17] Android Malware Genome Project - http://www.malgenomeproject.org/, Accessed on January 2015.

[18] Drebin Android Malware Dataset - https://user.informatik.uni-goettingen.de/~darp/drebin/, Accessed on January 2015.

[19] The Android Native Development Kit (NDK) - https://developer.android.com/ndk/index.html, Accessed on January 2016.

[20] Tracemyip - http://tools.tracemyip.org/, Accessed on January 2016.

[21] Contagiominidump Malware Repository - https://contagiominidump.blogspot.ca, Accessed on January 2017.

[22] HummingBad Android Malware Found in 20 Google Play Store Apps - https://www.bleepingcomputer.com/news/security/hummingbad-android-malware-found-in-20-google-play-store-apps/, Accessed on January 2017.

[23] Mumayi Market - http://www.mumayi.com/, Accessed on January 2017.

[24] Playdrone Android Dataset - https://archive.org/details/playdrone-apks, Accessed on January 2017.

[25] RaspberryPI 2 - https://www.raspberrypi.org/products/raspberry-pi-2-model-b/, Accessed on January 2017.

[26] Tensorflow - https://www.tensorflow.org, Accessed on January 2017.

[27] Android operating system - https://www.android.com/, Accessed on January 2019.

[28] Google Play - https://play.google.com/, Accessed on January 2020.

[29] Malware Statistics and Trends Report, AV-TEST - https://www.av-test.org/en/statistics/malware, Accessed on January 2020.

[30] 'Ransomware-as-a-service' discovered on the darknet - https://www.theguardian.com/technology/2015/jun/02/ransomware-as-service-discovered-on-darknet, Accessed on June 2015.

[31] 'Petya' ransomware attack: what is it and how can it be stopped? - https://www.theguardian.com/technology/2017/jun/27/petya-ransomware-cyber-attack-who-what-why-how, Accessed on June 2017.

[32] What is WannaCry and how does ransomware work? - http://www.telegraph.co.uk/technology/0/ransomware-does-work/, Accessed on June 2017.

[33] Android Software Developer Kit (SDK) - https://developer.android.com/studio/, Accessed on June 2019.

[34] Angr Framework - https://angr.io/, Accessed on June 2019.

[35] VEX Intermediate Representation - https://docs.angr.io/advanced-topics/ir, Accessed on June 2019.

[36] Report: Average of 82,000 new malware threats per day in 2013 - https://www.pcworld.com/article/2109210/report-average-of-82-000-new-malware-threats-per-day-in-2013.html, Accessed on March 2014.

[37] New Ransomware Encrypts Your Game Files - https://techcrunch.com/2015/03/24/new-ransomware-encrypts-your-game-files/, Accessed on March 2015.

[38] Android Wear Operating System - https://www.android.com/wear/, Accessed on March 2016.

[39] Android Platform Architecture - https://developer.android.com/guide/platform/index.html, Accessed on March 2017.

[40] Appchine Market - http://www.appchina.com/, Accessed on March 2017.

[41] New Malware Variants Near Record-Highs: Symantec - http://www.securityweek.com/new-malware-variants-near-record-highs-symantec, Accessed on March 2017.

[42] Six days after a ransomware cyberattack, atlanta officials are filling out forms by hand - https://edition.cnn.com/2018/03/27/us/atlanta-ransomware-computers/index.html, Accessed on March 2018.

[43] The Leaked NSA Spy Tool That Hacked the World - https://www.wired.com/story/eternalblue-leaked-nsa-spy-tool-hacked-world/, Accessed on March 2018.

[44] Android Things operating system - https://developer.android.com/things, Accessed on March 2020.

[45] Smartphone OS global market share https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/, Accessed on March 2020.

[46] Andr/Boxer-C malware family - https://www.sophos.com/en-us/threat-center/threat-analyses/viruses-and-spyware/Andr~Boxer-C.aspx, Accessed on May 2016.

[47] DroidBox - https://github.com/pjlantz/droidboxh, Accessed on May 2016.

[48] MonkeyRunner - https://developer.android.com/studio/test/monkeyrunner/index.html, Accessed on May 2016.

[49] Petya ransomware encryption system cracked - http://www.bbc.com/news/technology-36014810, Accessed on May 2016.

[50] Chimera Ransomware Promises to Publish Encrypted Data Online - https://threatpost.com/chimera-ransomware-promises-to-publish-\encrypted-data-online/115293/, Accessed on November 2015.

[51] Alexa Top Sites - http://www.alexa.com/topsites, Accessed on November 2016.

[52] Amazon IP Space - https://docs.aws.amazon.com/general/latest/gr/aws-ip-ranges.html, Accessed on November 2016.

[53] Quantcast Sites - https://tinyurl.com/gmd577y, Accessed on November 2016.

[54] Cyber attacks on Android devices on the rise - https://www.gdatasoftware.com/blog/2018/11/31255-cyber-attacks-on-android-devices-on-the-rise, Accessed on November 2018.

[55] New IoT Botnet Malware Discovered; Infecting More Devices Worldwide - http://thehackernews.com/2016/10/linux-irc-iot-botnet.html, Accessed on October 2016.

[56] Symantec Intelligence for February 2017 - https://www.symantec.com/connect/blogs/latest-intelligence-february-2017, Accessed on October 2017.

[57] Aggressive Android ransomware spreading in the USA - https://www.welivesecurity.com/2015/09/10/aggressive-android-ransomware-spreading-in-the-usa/, Accessed on September 2015.

[58] Android Things - https://developer.android.com/things/, Accessed on September 2016.

[59] Alert: 'Ryuk' Ransomware Attacks the Latest Threat - https://www.bankinfosecurity.com/alert-ryuk-ransomware-attacks-latest\-threat-a-11475, Accessed on September 2018.

[60] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *Security and Privacy in Communication Networks - 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers*, pages 86–103, 2013.

[61] Tony Abou-Assaleh, Nick Cercone, Vlado Keselj, and Ray Sweidan. N-gram-based detection of new malicious code. In *28th International Computer Software and Applications Conference (COMPSAC 2004), Design and Assessment of Trustworthy Software-Based Systems, 27-30 September 2004, Hong Kong, China, Workshop Papers*, pages 41–42, 2004.

[62] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy, CODASPY 2016, New Orleans, LA, USA, March 9-11, 2016*, pages 183–194, 2016.

[63] Aisha I. Ali-Gombe, Irfan Ahmed, Golden G. Richard III, and Vassil Roussev. Opseq: Android malware fingerprinting. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop, PPREW@ACSAC, Los Angeles, CA, USA, December 8, 2015*, pages 7:1–7:12, 2015.

[64] Aisha I. Ali-Gombe, Irfan Ahmed, Golden G. Richard III, and Vassil Roussev. Aspectdroid: Android app analysis system. In *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy, CODASPY 2016, New Orleans, LA, USA, March 9-11, 2016*, pages 145–147, 2016.

[65] Aisha I. Ali-Gombe, Brendan Saltaformaggio, J. Ramanujam, Dongyan Xu, and Golden G. Richard III. Toward a more dependable hybrid analysis of android malware using aspect-oriented programming. *Computers & Security*, 73:235–248, 2018.

[66] Joey Allen, Matthew Landen, Sanya Chaba, Yang Ji, Simon Pak Ho Chung, and Wenke Lee. Improving accuracy of android malware detection with lightweight contextual awareness. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 210–221, 2018.

[67] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, pages 468–471, 2016.

[68] Hussain M. J. Almohri, Danfeng (Daphne) Yao, and Dennis G. Kafura. Droidbarrier: know what is executing on your android. In *Fourth ACM Conference on Data and Application Security and Privacy, CODASPY'14, San Antonio, TX, USA - March 03 - 05, 2014*, pages 257–264, 2014.

[69] Saed Alrabaee, ElMouatez Billah Karbab, Lingyu Wang, and Mourad Debbabi. Bineye: Towards efficient binary authorship characterization using deep learning. In *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part II*, pages 47–67, 2019.

[70] Mohammed K. Alzaylaee, Suleiman Y. Yerima, and Sakir Sezer. Dynalog: An automated dynamic analysis framework for characterizing android applications. *CoRR*, abs/1607.08166, 2016.

[71] Brandon Amos, Hamilton A. Turner, and Jules White. Applying machine learning classifiers to dynamic android malware detection at scale. In *2013 9th International Wireless Communications and Mobile Computing Conference, IWCMC 2013, Sardinia, Italy, July 1-5, 2013*, pages 1666–1671, 2013.

[72] Manos Antonakakis, Roberto Perdisci, David Dagon, Wenke Lee, and Nick Feamster. Building a dynamic reputation system for dns. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, pages 273–290, 2010.

[73] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. DREBIN: effective and explainable detection of android malware in your pocket. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[74] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269, 2014.

[75] Michael Backes, Sven Bugiel, Erik Derr, Patrick D. McDaniel, Damien Octeau, and Sebastian Weisgerber. On demystifying the android application framework: Re-visiting android permission specification analysis. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 1101–1118, 2016.

[76] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. Appguard - fine-grained policy enforcement for untrusted android applications. In *Data Privacy Management and Autonomous Spontaneous Security - 8th International Workshop, DPM 2013, and 6th International Workshop, SETOP 2013, Egham, UK, September 12-13, 2013, Revised Selected Papers*, pages 213–231. 2013.

[77] Shikha Badhani and Sunil K. Muttoo. Cendroid - A cluster-ensemble classifier for detecting malicious android applications. *Computers & Security*, 85:25–40, 2019.

[78] David Barrera, Hilmi Günes Kayacik, Paul C. van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 73–84, 2010.

[79] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP 2012, Beijing, China, June 14, 2012*, pages 27–38, 2012.

[80] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. LSH forest: self-tuning indexes for similarity search. In *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*, pages 651–660, 2005.

[81] Shweta Bhandari, Rishabh Gupta, Vijay Laxmi, Manoj Singh Gaur, Akka Zemmari, and Maxim Anikeev. DRACO: droid analyst combo an android malware analysis framework. In *Proceedings of the 8th International Conference on Security of Information and Networks, SIN 2015, Sochi, Russian Federation, September 8-10, 2015*, pages 283–289, 2015.

[82] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *J. Stat. Mech. Theory Exp.*, 2008.

[83] Stephen P. Borgatti. Centrality and network flow. *Social Networks*, 27(1):55–71, 2005.

[84] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[85] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *SPSM'11, Proceedings of the 1st ACM Workshop Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2011, October 17, 2011, Chicago, IL, USA*, pages 15–26, 2011.

[86] Pete Burnap, Richard French, Frederick Turner, and Kevin Jones. Malware classification using self organising feature maps and machine activity data. *Computers & Security*, 73:399–410, 2018.

[87] Gerardo Canfora, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Acquiring and analyzing app metrics for effective mobile malware detection. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics, IWSPA@CODASPY 2016, New Orleans, LA, USA, March 11, 2016*, pages 50–57, 2016.

[88] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 659–674, 2015.

[89] Li Chen, Mingwei Zhang, Chih-Yuan Yang, and Ravi Sahita. POSTER: semi-supervised classification for dynamic android malware detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2479–2481, 2017.

[90] Qian Chen and Robert A. Bridges. Automated behavioral analysis of malware: A case study of wannacry ransomware. In *16th IEEE International Conference on Machine Learning and Applications, ICMLA 2017, Cancun, Mexico, December 18-21, 2017*, pages 454–460, 2017.

[91] Sen Chen, Minhui Xue, Zhushou Tang, Lihua Xu, and Haojin Zhu. Stormdroid: A streaminglized machine learning-based system for detecting android malware. In *Proceedings of the*

11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016, pages 377–388, 2016.

[92] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David A. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys 2011), Bethesda, MD, USA, June 28 - July 01, 2011*, pages 239–252, 2011.

[93] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barenghi, Stefano Zanero, and Federico Maggi. Shieldfs: a self-healing, ransomware-aware filesystem. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, pages 336–347, 2016.

[94] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, pages 37–54, 2012.

[95] Jonathan Crussell, Clint Gibler, and Hao Chen. Andarwin: Scalable detection of semantically similar android applications. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, pages 182–199, 2013.

[96] Jonathan Crussell, Clint Gibler, and Hao Chen. Andarwin: Scalable detection of android application clones based on semantics. *IEEE Trans. Mob. Comput.*, 14(10):2007–2019, 2015.

[97] Dimitrios Damopoulos, Georgios Kambourakis, and Georgios Portokalidis. The best of both worlds: a framework for the synergistic operation of host and cloud anomaly-based IDS for smartphones. In *Proceedings of the Seventh European Workshop on System Security, EuroSec 2014, April 13, 2014, Amsterdam, The Netherlands*, pages 6:1–6:6, 2014.

[98] Franca Delmastro, Valerio Arnaboldi, and Marco Conti. People-centric computing and communications in smart cities. *IEEE Communications Magazine*, 54(7):122–128, 2016.

[99] Luke Deshotels, Vivek Notani, and Arun Lakhotia. Droidlegacy: Automated familial classification of android malware. In *Proceedings of the 3rd ACM SIGPLAN Program Protection and Reverse Engineering Workshop 2014, PPREW 2014, January 25, 2014, San Diego, CA, USA*, pages 3:1–3:12, 2014.

[100] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 472–489, 2019.

[101] Karim O. Elish, Xiaokui Shu, Danfeng (Daphne) Yao, Barbara G. Ryder, and Xuxian Jiang. Profiling user-trigger dependence for android malware detection. *Computers & Security*, 49:255–273, 2015.

[102] Karim O Elish, Danfeng Yao, and Barbara G Ryder. User-centric dependence analysis for identifying malicious mobile apps. In *2012 IEEE Security and Privacy Workshops, SP Workshops 2012*, 2016.

[103] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick D. McDaniel, and Anmol Sheth. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Commun. ACM*, 57(3):99–106, 2014.

[104] William Enck, Machigar Ongtang, and Patrick D. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 235–245, 2009.

[105] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Zhenzhou Tian, Qinghua Zheng, and Ting Liu. Android malware familial classification and representative sample selection via frequent subgraph analysis. *IEEE Trans. Information Forensics and Security*, 13(8):1890–1905, 2018.

[106] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. Android security: A survey of issues, malware penetration, and defenses. *IEEE Communications Surveys and Tutorials*, 17(2):998–1022, 2015.

[107] Parvez Faruki, Vijay Ganmoor, Vijay Laxmi, Manoj Singh Gaur, and Ammar Bharmal. Androsimilar: robust statistical feature signature for android malware detection. In *The 6th International Conference on Security of Information and Networks, SIN '13, Aksaray, Turkey, November 26-28, 2013*, pages 152–159, 2013.

[108] Parvez Faruki, Vijay Laxmi, Ammar Bharmal, Manoj Singh Gaur, and Vijay Ganmoor. Androsimilar: Robust signature for detecting variants of android malware. *J. Inf. Sec. Appl.*, 22:66–80, 2015.

[109] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David A. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 627–638, 2011.

[110] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David A. Wagner. Android permissions: user attention, comprehension, and behavior. In *Symposium On Usable Privacy and Security, SOUPS '12, Washington, DC, USA - July 11 - 13, 2012*, page 3, 2012.

[111] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 576–587, 2014.

[112] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. Scandroid: Automated security certification of android. 2009. `https://www.cs.umd.edu/~avik/papers/scandroidascaa.pdf`.

[113] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[114] Michael C. Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker:

scalable and accurate zero-day android malware detection. In *The 10th International Conference on Mobile Systems, Applications, and Services, MobiSys'12, Ambleside, United Kingdom - June 25 - 29, 2012*, pages 281–294, 2012.

[115] Steve Hanna, Ling Huang, Edward XueJun Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 9th International Conference, DIMVA 2012, Heraklion, Crete, Greece, July 26-27, 2012, Revised Selected Papers*, pages 62–81. 2012.

[116] Geoffrey E. Hinton, Alex Krizhevsky, and Sida D. Wang. Transforming auto-encoders. In *Artificial Neural Networks and Machine Learning - ICANN 2011 - 21st International Conference on Artificial Neural Networks, Espoo, Finland, June 14-17, 2011, Proceedings, Part I*, pages 44–51, 2011.

[117] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[118] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. Asdroid: detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 1036–1046, 2014.

[119] Fauzia Idrees, Muttukrishnan Rajarajan, Mauro Conti, Thomas M. Chen, and Yogachandran Rahulamathavan. Pindroid: A novel android malware detection system using ensemble learning methods. *Computers & Security*, 68:36–46, 2017.

[120] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 448–456, 2015.

[121] Jae-wook Jang, Hyun-Jae Kang, Jiyoung Woo, Aziz Mohaisen, and Huy Kang Kim. Andro-dumpsys: Anti-malware system based on the similarity of malware creator and malware centric information. *Computers & Security*, 58:125–138, 2016.

[122] Jiyong Jang, David Brumley, and Shobha Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 309–320, 2011.

[123] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A Hidden Code Extractor for Packed Executables. *ACM workshop on Recurring malcode (WORM)*, 2007.

[124] ElMouatez Billah Karbab and Mourad Debbabi. Automatic investigation framework for android malware cyber-infrastructures. *CoRR*, abs/1806.08893, 2018.

[125] ElMouatez Billah Karbab and Mourad Debbabi. Portable, data-driven malware detection using language processing and machine learning techniques on behavioral analysis reports. *CoRR*, abs/1812.10327, 2018.

[126] ElMouatez Billah Karbab and Mourad Debbabi. Togather: Automatic investigation of android malware cyber-infrastructures. In *Proceedings of the 13th International Conference on Availability, Reliability and Security, ARES 2018, Hamburg, Germany, August 27-30, 2018*, pages 20:1–20:10, 2018.

[127] ElMouatez Billah Karbab and Mourad Debbabi. Maldy: Portable, data-driven malware detection using natural language processing and machine learning techniques on behavioral analysis reports. *Digital Investigation*, 28(Supplement):S77–S87, 2019.

[128] ElMouatez Billah Karbab, Mourad Debbabi, Saed Alrabaee, and Djedjiga Mouheb. Dysign: dynamic fingerprinting for the automatic detection of android malware. In *11th International Conference on Malicious and Unwanted Software, MALWARE 2016, Fajardo, PR, USA, October 18-21, 2016*, pages 139–146, 2016.

[129] ElMouatez Billah Karbab, Mourad Debbabi, Saed Alrabaee, and Djedjiga Mouheb. Dysign: Dynamic fingerprinting for the automatic detection of android malware. *CoRR*, abs/1702.05699, 2017.

[130] ElMouatez Billah Karbab, Mourad Debbabi, Abdelouahid Derhab, and Djedjiga Mouheb. Cypider: building community-based cyber-defense infrastructure for android malware detection. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, pages 348–362, 2016.

[131] ElMouatez Billah Karbab, Mourad Debbabi, Abdelouahid Derhab, and Djedjiga Mouheb. Android malware detection using deep learning on API method sequences. *CoRR*, abs/1712.08996, 2017.

[132] ElMouatez Billah Karbab, Mourad Debbabi, Abdelouahid Derhab, and Djedjiga Mouheb. Maldozer: Automatic framework for android malware detection using deep learning. *Digital Investigation*, 24:S48–S59, 2018.

[133] ElMouatez Billah Karbab, Mourad Debbabi, and Djedjiga Mouheb. Fingerprinting Android packaging: Generating DNAs for malware detection. *Digital Investigation*, 18:S33—-S45, 2016.

[134] Amin Kharraz, Sajjad Arshad, Collin Mulliner, William K. Robertson, and Engin Kirda. UNVEIL: A large-scale, automated approach to detecting ransomware. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 757–772, 2016.

[135] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, Junbum Shin, and SWRD Center. (POSTER) ScanDal: Static analyzer for detecting privacy leaks in android applications. *IEEE Security and Privacy*, 12, 2012.

[136] Junhyoung Kim, TaeGuen Kim, and Eul Gyu Im. Structural information based malicious app similarity calculation and clustering. In *Proceedings of the 2015 Conference on research in adaptive and convergent systems, RACS 2015, Prague, Czech Republic, October 9-12, 2015*, pages 314–318, 2015.

[137] TaeGuen Kim, BooJoong Kang, Mina Rho, Sakir Sezer, and Eul Gyu Im. A multimodal deep learning method for android malware detection using various features. *IEEE Trans. Information Forensics and Security*, 14(3):773–788, 2019.

[138] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1746–1751, 2014.

[139] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[140] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.

[141] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. pages 351–366, 2009.

[142] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[143] Jehyun Lee, Suyeon Lee, and Heejo Lee. Screening smartphone applications using malware family signatures. *Computers & Security*, 52:234–249, 2015.

[144] Ying-Dar Lin, Yuan-Cheng Lai, Chien-Hung Chen, and Hao-Chuan Tsai. Identifying android malicious repackaged applications by thread-grained system call sequences. *Computers & Security*, 39:340–350, 2013.

[145] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. ANDRUBIS - 1, 000, 000 apps later: A view on current android malware behaviors. In *Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security, BADGERS@ESORICS 2014, Wroclaw, Poland, September 11, 2014*, pages 3–17, 2014.

[146] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. volume 5, pages 40–45, 2007.

[147] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.

[148] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon J. Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[149] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *23rd Annual Computer Security Applications Conference (ACSAC 2007), December 10-14, 2007, Miami Beach, Florida, USA*, pages 431–441, 2007.

[150] Fabio Martinelli, Francesco Mercaldo, and Andrea Saracino. BRIDEMAID: an hybrid tool for accurate detection of android malware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, pages 899–901, 2017.

[151] Mohammad M. Masud, Latifur Khan, and Bhavani M. Thuraisingham. A scalable multi-level feature extraction technique to detect malicious executables. *Information Systems Frontiers*, 10(1):33–45, 2008.

[152] Niall McLaughlin, Jesús Martínez del Rincón, BooJoong Kang, Suleiman Y. Yerima, Paul C. Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. Deep android malware detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017*, pages 301–308, 2017.

[153] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed

representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, pages 3111–3119, 2013.

[154] Ritika Wason Nidhi Grover. Comparative Analysis Of Pagerank And HITS Algorithms. *Int. J. Eng. Res. Technol.*, 2012.

[155] Jon Oberheide and Charlie Miller. Dissecting the android bouncer. *SummerCon2012, New York*, 2012.

[156] Philip O'Kane, Sakir Sezer, and Kieran McLaughlin. Obfuscation: The hidden malware. *IEEE Security & Privacy*, 9(5):41–47, 2011.

[157] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon J. Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). *ACM Trans. Priv. Secur.*, 22(2):14:1–14:34, 2019.

[158] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon J. Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). *ACM Trans. Priv. Secur.*, 22(2):14:1–14:34, 2019.

[159] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543, 2014.

[160] Bahman Rashidi, Carol J. Fung, and Elisa Bertino. Android resource usage risk assessment using hidden markov model and online learning. *Computers & Security*, 65:90–107, 2017.

[161] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *21st Annual Network and Distributed*

System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014, 2014.

[162] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, pages 329–334, 2013.

[163] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April*, 2013.

[164] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.

[165] Andrew Rosenberg and Julia Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *EMNLP-CoNLL 2007, Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, June 28-30, 2007, Prague, Czech Republic*, pages 410–420, 2007.

[166] Justin Sahs and Latifur Khan. A machine learning approach to android malware detection. In *2012 European Intelligence and Security Informatics Conference, EISIC 2012, Odense, Denmark, August 22-24, 2012*, pages 141–147, 2012.

[167] Borja Sanz, Igor Santos, Xabier Ugarte-Pedrero, Carlos Laorden, Javier Nieves, and Pablo García Bringas. Anomaly detection using string analysis for android malware detection. In *International Joint Conference SOCO'13-CISIS'13-ICEUTE'13 - Salamanca, Spain, September 11th-13th, 2013 Proceedings*, pages 469–478, 2013.

[168] Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli. MADAM: effective and efficient behavior-based android malware detection and prevention. *IEEE Trans. Dependable Sec. Comput.*, 15(1):83–97, 2018.

[169] Bhaskar Pratim Sarma, Ninghui Li, Christopher S. Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Android permissions: a perspective combining risks and benefits. In *17th ACM Symposium on Access Control Models and Technologies, SACMAT '12, Newark, NJ, USA - June 20 - 22, 2012*, pages 13–22, 2012.

[170] Bernhard Schölkopf, John C. Platt, John Shawe-Taylor, Alexander J. Smola, and Robert C. Williamson. Estimating the support of a high-dimensional distribution. *Neural Computation*, 13(7):1443–1471, 2001.

[171] Daniele Sgandurra, Luis Muñoz-González, Rabih Mohsen, and Emil C. Lupu. Automated dynamic analysis of ransomware: Benefits, limitations and use for detection. *CoRR*, abs/1609.03020, 2016.

[172] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. "andromaly": a behavioral malware detection framework for android devices. *J. Intell. Inf. Syst.*, 38(1):161–190, 2012.

[173] Qinfeng Shi, James Petterson, Gideon Dror, John Langford, Alexander J. Smola, and S. V. N. Vishwanathan. Hash kernels for structured data. volume 10, pages 2615–2637, 2009.

[174] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Visual Recognition - `http://www.robots.ox.ac.uk/~vgg/research/very_deep/`, Accessed on January 2017.

[175] Michael Spreitzenbarth, Felix C. Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, pages 1808–1815, 2013.

[176] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, 2014.

[177] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. Droidsieve: Fast and accurate classification of obfuscated android malware. pages 309–320, 2017.

[178] Guillermo Suarez-Tangil, Juan E. Tapiador, Pedro Peris-Lopez, and Jorge Blasco Alís. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Syst. Appl.*, 41(4):1104–1117, 2014.

[179] Sufatrio, Darell J. J. Tan, Tong-Wei Chua, and Vrizlynn L. L. Thing. Securing android: A survey, taxonomy, and challenges. *ACM Comput. Surv.*, 47(4):58:1–58:45, 2015.

[180] Mingshen Sun, Mengmeng Li, and John C. S. Lui. Droideagle: seamless detection of visually similar android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, New York, NY, USA, June 22-26, 2015*, pages 9:1–9:12, 2015.

[181] Mingshen Sun, Xiaolei Li, John C.S. S Lui, Richard T.B. B Ma, and Zhenkai Liang. Monet: A User-Oriented Behavior-Based Malware Variants Detection System for Android. *IEEE Transactions on Information Forensics and Security*, 12(5):1103–1112, 2017.

[182] Mingshen Sun, Min Zheng, John C. S. Lui, and Xuxian Jiang. Design and implementation of an android host-based intrusion prevention system. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, pages 226–235, 2014.

[183] Ke Tian, Danfeng Daphne Yao, Barbara G Ryder, Gang Tan, and Guojun Peng. Detection of repackaged android malware with code-heterogeneity features. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[184] Timothy Vidas, Jiaqi Tan, Jay Nahata, Chaur Lih Tan, Nicolas Christin, and Patrick Tague. A5: automated analysis of adversarial android applications. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, SPSM@CCS 2014, Scottsdale, AZ, USA, November 03 - 07, 2014*, pages 39–50, 2014.

[185] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*, pages 252–276, 2017.

[186] Te-En Wei, Ching-Hao Mao, Albert B. Jeng, Hahn-Ming Lee, Horng-Tzer Wang, and Dong-Jie Wu. Android malware detection via a latent network behavior analysis. In *11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2012, Liverpool, United Kingdom, June 25-27, 2012*, pages 1251–1258, 2012.

[187] Florian Weimer. Passive DNS Replication, 2005. `http://www.enyo.de/fw/software/dnslogger/first2005-paper.pdf`.

[188] Kilian Q. Weinberger, Anirban Dasgupta, John Langford, Alexander J. Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, pages 1113–1120, 2009.

[189] Carsten Willems, Thorsten Holz, and Felix C. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, 5(2):32–39, 2007.

[190] Ho Chung Wu, Robert Wing Pong Luk, Kam-Fai Wong, and Kui-Lam Kwok. Interpreting TF-IDF term weights as making relevance decisions. *ACM Trans. Inf. Syst.*, 26(3):13:1–13:37, 2008.

[191] Ke Xu, Yingjiu Li, and Robert H. Deng. Iccdetector: Icc-based malware detection on android. *IEEE Trans. Information Forensics and Security*, 11(6):1252–1264, 2016.

[192] Ke Xu, Yingjiu Li, Robert H. Deng, and Kai Chen. Deeprefiner: Multi-layer android malware detection system applying deep neural networks. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 473–487, 2018.

[193] Ke Xu, Yingjiu Li, Robert H. Deng, and Kai Chen. Deeprefiner: Multi-layer android malware detection system applying deep neural networks. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 473–487, 2018.

[194] Ke Xu, Yingjiu Li, Robert H. Deng, Kai Chen, and Jiayun Xu. Droidevolver: Self-evolving android malware detection system. In *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*, pages 47–62, 2019.

[195] Rubin Xu, Hassen Saïdi, and Ross J. Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 539–552, 2012.

[196] Wenbo Yang, Juanru Li, Yuanyuan Zhang, Yong Li, Junliang Shu, and Dawu Gu. Apklancet: tumor payload diagnosis and purification for android applications. In *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, pages 483–494, 2014.

[197] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual API dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1105–1116, 2014.

[198] Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 649–657, 2015.

[199] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, Xiaoyang Sean Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 611–622, 2013.

[200] Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie. Droidalarm: an all-sided static analysis tool for android privilege-escalation malware. In *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, pages 353–358, 2013.

[201] Wu Zhou, Yajin Zhou, Michael C. Grace, Xuxian Jiang, and Shihong Zou. Fast, scalable detection of "piggybacked" mobile applications. In *Third ACM Conference on Data and Application Security and Privacy, CODASPY'13, San Antonio, TX, USA, February 18-20, 2013*, pages 185–196, 2013.

[202] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Second ACM Conference on Data and Application Security and Privacy, CODASPY 2012, San Antonio, TX, USA, February 7-9, 2012*, pages 317–326, 2012.

[203] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 95–109, 2012.

[204] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.